Scheduling Time-Constrained Instructions on Pipelined Processors

ALLEN LEUNG
New York University
and
KRISHNA V. PALEM
Georgia Institute of Technology and New York University
and
AMIR PNUELI
Weizmann Institute of Science and New York University

In this work we investigate the problem of scheduling instructions on RISC and VLIW machines with multiple pipelines, in the presence of precedence constraints, release-times, deadlines, and latencies constraints. A latency of l_{ij} specifies that there must be at least l_{ij} time-steps between the completion time of instruction i and the start time of instruction j. Latency of $l_{ij} = -1$ can be used to specify that j may be scheduled concurrently with i but not earlier. We present a generic algorithm that runs in $O(n^2 \log n\alpha(n) + ne)$ time, given n instructions and e edges in the precedence DAG, where $\alpha(n)$ is the functional inverse of the Ackermann function. Our algorithm can be used to construct feasible schedules for various classes of instances, including instances with the following configurations: (1) one pipeline, with individual release-times and deadlines and where the latencies between instructions are restricted to 0 and 1; (2) m pipelines, with individual release-times and deadlines, and monotone-interval order precedences; (3) two pipelines with latencies of -1 or 0, and release-times and deadlines; (4) one pipeline, latencies of 0 or 1 and individual processing times that are at least one; (5) m pipelines, intree precedences, constant latencies, and deadlines; (6) m pipelines, outtree precedences, constant latencies, and release-times. For instances with deadlines, optimal schedules that minimize the maximal tardiness can be constructed using binary search, in $O(\log n)$ iterations of our algorithm. We obtain our results using backward scheduling, a very general relaxation method, which extends, unifies and clarifies many previous results on instruction scheduling for pipelined and parallel machines.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*code generation; optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*

General Terms: Algorithms

Additional Key Words and Phrases: Compiler optimization, deadlines, instruction scheduling, latency, NP-complete, pipeline processor, precedence delay, real-time, release-times, RISC machine scheduling, time lag

Supported in part by awards from the Hewlett-Packard Corporation, Panasonic American Laboratories, the IBM corporation and DARPA contract no. DABT63-96-C-0049.

1. INTRODUCTION

Instruction scheduling, a key phase in most optimizing compilers, is an optimization that exploits *instruction level parallelism* (ILP) in modern RISC and VLIW processors. Instruction scheduling algorithms typically attempt to minimize the overall length, or *makespan*, of a given set of instructions. The core case of instruction-scheduling for basic-blocks is now a very well-understood subject. Provably optimal solutions have been found for some idealized machine models, and many efficient heuristics have been validated in product-quality compilers [5; 21; 34; 43; 16]. By extending these basic techniques, more advanced scheduling algorithms that exploit parallelism at a global level and between loop iterations have been developed [6; 32; 16; 24; 27; 37; 38].

Recently, with their ever-decreasing cost and improving performance, modern RISC processors are being used more and more as controllers of automobile braking systems, monitors of intensive care rooms in hospitals, telemetry and a variety of other embedded applications. In addition to completing the instructions rapidly via makespan minimization, embedded applications utilizing these processors often require that specific timing relationships to hold between the instructions. For example, some of the instructions might have tight time constraints specified as *deadlines* and *release-times*. Deadlines denote upper-bounds on the times by which instructions must complete, whereas release-times denote lower-bounds on the start time of the instructions in a schedule.

Aside from embedded applications, time-constraints also appear naturally in numerous situations in instruction scheduling that extend beyond basic blocks. For example, region-based global scheduling algorithms [20] may require careful "meshing" of two scheduled regions, so that their schedules fit well together. Latencies corresponding to data-dependences that cross region boundaries can naturally be translated into the appropriate release-times and deadlines in two regions.

In this paper, we describe the first polynomial time algorithm[1] that is guaranteed to find feasible schedules given basic-blocks of code with these time constraints, whenever such schedules exist, on various models of RISC machines. In doing so, we unify and generalize many earlier results on scheduling for pipelined machines, including the earlier work due to Palem and Simons [34] on scheduling on pipelined machines with deadlines, and that of Garey and Johnson on scheduling tasks with release-times and deadlines on two identical processors [14], plus other related pipelined scheduling problems with identical latencies proposed by Bruno, Jones and So [8].

Our algorithm runs in $O(n^2 \log n\alpha(n) + ne)$ time[2], where n is the number of instructions to be scheduled, and e is the number of edges in the input DAG, and is guaranteed to find feasible schedules for arbitrary basic-blocks of code, for such early RISC machines as the IBM-801 [36], the Berkeley RISC [26] and Stanford MIPS [22] processors. Also, our algorithm can in the same time bound find feasible schedules whenever such schedules exist, for basic-blocks whose data-dependence graphs are *monotone interval-orders* [34; 35].

This rest of the paper is structured as follows. In Section 2 we provide the for-

[1] Some of the results in this paper appeared previously in our conference paper [30].
[2] $\alpha(n)$ is the functional inverse of the Ackermann function.

mal definition of the optimization/decision questions, and discuss related modeling issues. In Section 3, we relate our work with previous research in the rich area of deterministic scheduling. Section 4 describes our scheduling framework, and introduces the notion of a *backward schedule*, a specific form of relaxation used for computing *modified deadlines*. Our algorithm computes feasible schedules by list-scheduling the instructions in non-decreasing order of their modified deadlines. In Section 5 we show that for many problem classes of interest, schedules produced from our algorithm are always feasible, whenever such schedules exist.

A crucial technical idea on which our algorithm rests is that of a *stable* and *consistent* modified deadline associated with each instruction. The major algorithmic contribution in this work is a fast convergence scheme for computing the modified deadlines. As shown in Section 4, it is quite easy to compute the modified deadlines of a given input DAG in n^2 invocations of a key backward-scheduling primitive. The challenge is in showing that stability and consistency can be preserved and that a more sophisticated, yet simple scheme can achieve the same goal in just $O(n)$ iterations. An efficient implementation of the scheme is presented in Section 6. Finally, in Section 7 we consider the situation where processing times of instructions can be greater than one.

2. MODELING AND DEFINITIONS

As in previous work, we assume that the basic-block being scheduled is represented as a DAG, with the instructions represented as nodes and the dependences between instructions represented as edges. An instruction i requires p_i cycles of *processing time* to execute on any of the m *pipelined* functional units in the processor. On RISC machines, processing times p_i's are used to model the number of cycles that a pipeline must wait before a new instruction can be issued, i.e. the number of *interlock* cycles, plus one. In this paper we are mainly concerned with *u*nit *e*xecution *t*ime (UET) models[3], where $p_i = 1$. This corresponds to models in which no instruction can stall the pipeline.

On RISC machines, results computed from one instruction may not be immediately available to subsequent instructions. For example, if an instruction i is a load instruction, there might be additional delays due to access latencies to memory. Therefore, an instruction j that is data-dependent on i might have to be delayed extra l_{ij} cycles after i completes, before it can start executing. Here, the value $l_{ij} \geq -1$ is referred to as the *latency* between i and j. Graphically the latency l_{ij} is often represented as the label associated with the edge (i, j) in the precedence DAG.

Note that in our model we assume that the latency is a function of both the producer and consumer instructions. This extra flexibility allows us to model machines with result forwarding hardware between restricted set of functional units. In these machines the edge latencies from the same source instruction may be different, depending on the type of producer functional unit and the consumer function unit.

Unlike previous work in this area [34; 7], we allow latencies of -1 in our model. Given two instructions i and j, $l_{ij} = -1$ can be interpreted to mean that j can be issued *concurrently* with i but not any earlier. Clearly, in the presence of only

[3]In Section 7 we will consider the cases with non-unit processing times.

one pipeline latency of -1 is equivalent to latency of 0. However, in the presence of multiple pipelines, this form of latency is very useful in practice since it can be used to encode *anti-dependence* introduced by the use of shared resources[4]. For example, if operation i precedes operation j in the linearized program, and i reads from some register r and j writes to the same, i and j can be issued concurrently in the same VLIW packet without violating the program semantics, since registers are read at the beginning of the execution cycle while results are written back at the end of the same cycle.

We use $\mathbf{I}$ to denote the set of instructions to be scheduled, where $|\mathbf{I}| = n$. Aside from latencies and resources constraints, the feasible schedule time of an instruction i can be constrained by release-times and deadlines, which we denote as r_i and d_i. We use the partial order relation $\prec$ to denote the precedence constraints on instructions, i.e. $i \prec j$, then instruction i must complete before j begins execution. The *predecessors* of i in a DAG is the set $pred_i \stackrel{\text{def}}{=} \{j \in \mathbf{I} \mid j \prec i\}$ and the *successors* of i is the set $succ_i \stackrel{\text{def}}{=} \{j \in \mathbf{I} \mid i \prec j\}$. If $i \not\prec j$ and $j \not\prec i$, we say that i and j are *independent* or incomparable. We denote the set of instructions independent of i as $indep_i \stackrel{\text{def}}{=} \{j \in \mathbf{I} \mid i \not\prec j \wedge j \not\prec i\}$.

The **Time-Constrained Pipelined Scheduling** (TCPS) problem that we consider in this paper can be phrased formally as follows:
INSTANCE: Given a set of instructions $\mathbf{I}$, a DAG that encodes the precedence order $\prec$, individual latencies $l : \mathbf{I} \times \mathbf{I} \rightharpoonup \mathbf{N}$, release-times $r : \mathbf{I} \to \mathbf{N}$, deadlines $d : \mathbf{I} \to \mathbf{N}$, and m identical pipelines.
QUESTION: Is there a schedule $\sigma : \mathbf{I} \to \mathbf{N}$ that satisfies the following precedence, resource, release-time and deadline constraints?

(1) **precedences and latencies**: for all edges $i \to j$ in the DAG, $\sigma(i) + l_{ij} + p_i \leq \sigma(j)$
(2) **resource**: for all time t, $|\{i \in \mathbf{I} \mid \sigma(i) \leq t < \sigma(i) + p_i\}| \leq m$
(3) **release-times**: for all $i \in \mathbf{I}$, $r_i \leq \sigma(i)$, and
(4) **deadlines**: for all $i \in \mathbf{I}$, $\sigma(i) + p_i \leq d_i$.

In the rest of the paper we will follow the $\alpha|\beta|\gamma$ notation of Graham et al. [17], augmented with latency notations introduced in Brucker and Knust [7], to describe scheduling problems. In this notation the field α defines the machine environment, such as one machine (1), multiple identical machines (P), or m identical machines (Pm). The field β represents constraints such as precedence constraints, latencies, processing times, and release times. The optimization criterion, such as makespan ($C_{\max}$), lateness ($L_{\max}$), or average completion time ($\sum C_{\max}$), is represented by γ. For instance, the TCPS problem can be denoted as $P|prec(l_{ij}); r_i|L_{\max}$.

Frequently we deal with restrictions on the form of precedence constraints. An *outtree*(*intree*) is a DAG where each instruction has no more than one immediate predecessor (successor). An *interval-order* graph is a DAG $G = (V, E)$ such that there exists a mapping f from V to intervals on the real line with the following

[4]In fact, in more modern VLIW machines, negative latencies are also useful for encoding other forms of dependence. This simple extension is not considered in this paper.

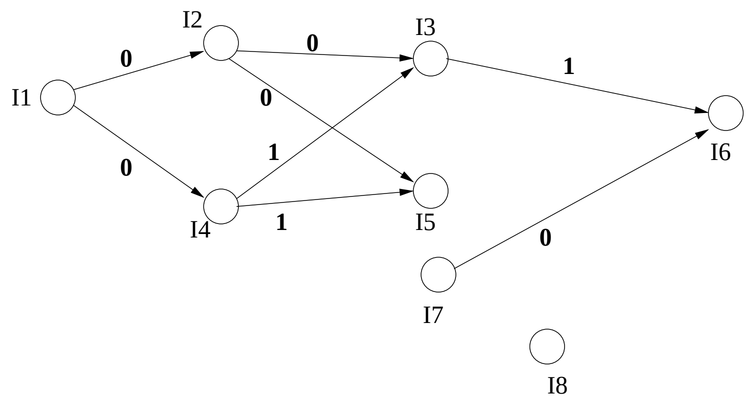

Fig. 1. An example DAG

Time	0	1	2	3	4	5	6	7
Instruction	$I1$	$I4$	$I2$	$I3$	$I5$	$I7$	$I6$	$I8$

Fig. 2. Optimal schedule S_1 in the absence of release-times and deadlines.

property: $(i,j) \in E$ iff $x \in f(i)$ and $y \in f(j)$ implies $x < y$. A *monotone* interval-order graph is an interval-order graph in which for every pair of edges $(i,j),(i,j') \in E$, $l_{ij} \leq l_{ij'}$ if $pred_j \subseteq pred_{j'}$. We will denote the TCPS problem restricted to monotone interval-orders as

$$P|int-order(mono\ l_{ij});r_i|L_{\max}$$

For convenience, we assume that the transitive latency l^+_{ij} between all pairs of nodes i and j is given as the input. If these values are not given, they can be computed within our stated time-bounds. For convenience, we assume $l^+_{ij} = -\infty$ if $i \not\prec j$.

2.1 An Example

Let us consider the example DAG in Figure 1. The latencies are shown as labels next to the edges. A schedule that satisfies the precedence, latency and resource constraints on a processor with a single pipeline ($m = 1$) is shown in Figure 2. Note that instruction $I3$ cannot be scheduled in time-step 2 due to its latency with instruction $I4$; it must wait an extra-cycle after $I4$ completes during step 2.

Suppose the following the additional timing constraints are imposed to the problem: release-times $r_2 = 1$, $r_7 = 6$, $r_8 = 6$ and deadlines $d_1 = 2$, $d_2 = 2$, $d_5 = 7$, $d_8 = 8$, and $d_j = 9$ for $j \in \{3,4,6,7\}$. In Figure 3 we show the same DAG annotated with these constraints; we represent release-time constraints as arrows to the left of a node, and deadline constraints as arrows to the right.

We say that instruction i meets its deadline if $\sigma(i) < d_i$ and misses its deadline if $\sigma(i) \geq d_i$. The schedule in Figure 2 is not feasible with these new constraints, since $I2$ misses its deadline. By analyzing the structure of the problem, we can infer that instruction $I2$ must be scheduled in time step 1 in *any* feasible schedule.

A feasible schedule S_2 for this problem is displayed in Figure 4. It also happens

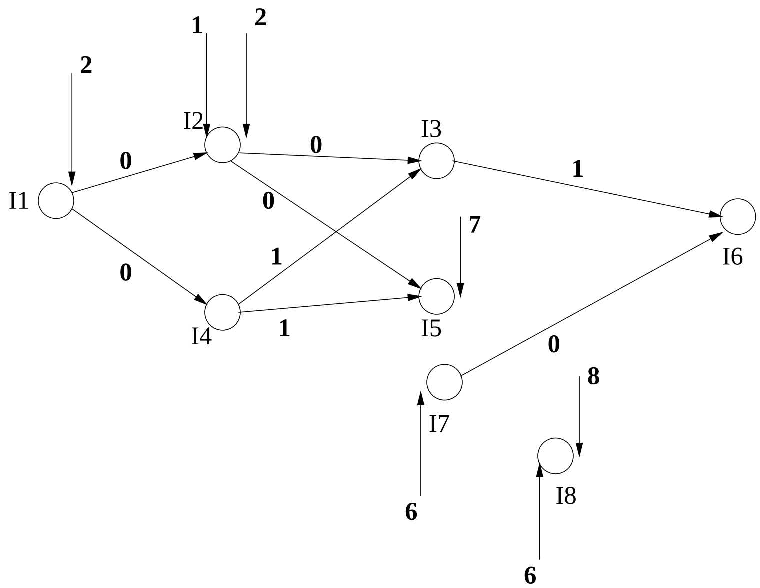

Fig. 3. An example DAG with release-times and deadlines.

Time	0	1	2	3	4	5	6	7	8
Instruction	$I1$	$I2$	$I4$		$I5$	$I3$	$I7$	$I8$	$I6$

Fig. 4. Feasible time-constrained schedule S_2.

to be optimal, i.e. it has the smallest overall completion time among all feasible schedules for this instance.

3. SUMMARY AND COMPARISONS

3.1 Polynomially Solvable Instances

The algorithm presented in this work is guaranteed to find a feasible schedule for unit-processing-time instances whenever such a schedule exists in $O(n^2 \log n\alpha(n) + ne)$ time, given

(1) one functional unit, arbitrary precedences as well as arbitrary release-times and deadlines, provided the latencies l_{ij} are either zero or one. These restrictions on the latencies model early RISC processors such as the IBM-801, Berkeley RISC and Stanford MIPS processors [21; 43].

(2) DAGs that are monotone interval-orders, on machines with an arbitrary number m of pipelines, with arbitrary release times and deadlines.

(3) two pipelines, arbitrary precedences, release-times, deadlines and latencies of -1 or 0.

(4) outtree precedences with release-times and constant latencies.

(5) intree precedences with deadlines-times and constant latencies.

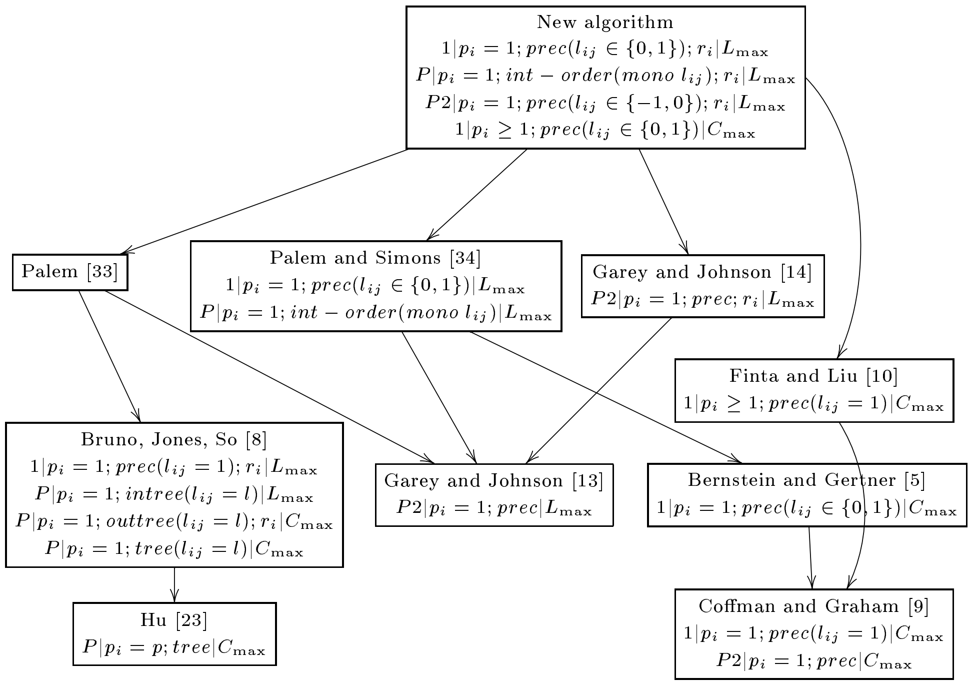

Fig. 5. Specializing our algorithm to known results. Our algorithm can solve all problems listed. Arrows denote generalization of results.

The running times of the algorithm can be improved to $O(n^2 \log n + ne)$ in all cases using the techniques identified by Gabow and Tarjan [12], and by Frederickson [11], using possibly more than $O(n^2)$ space[5].

To construct schedules with minimal tardiness, the standard technique of binary search can be used. This adds an extra logarithmic factor to the time complexity.

One interesting aspect of our result is that our algorithm also unifies many previously known results involving latencies, deadlines and release-times (see Figure 5.) We group these results into three parts.

(1) In the first group, we have a range of scheduling problems introduced by Bruno, Jones and So wherein all the latencies are *equal*, and possibly zero [8]; the latter case yields the well-known multiprocessor scheduling problem. Furthermore, we do not consider release-times; only deadlines are permitted. We derive this fact by specializing our algorithm to yield the framework described in Palem's thesis [33] which unified and generalized all the previously known results in this category.

(2) With possibly unequal latencies as is the case with RISC processors, we can specialize our algorithm to get polynomially solvable results due to Palem and Simons [34], as well as those due to Bernstein and Gertner [5] for deadline-free instances. Again, release-times are not considered. The former algorithm

[5]The algorithm can use space that is proportional to the difference between the values of the largest of all deadlines, and the smallest of all release-times.

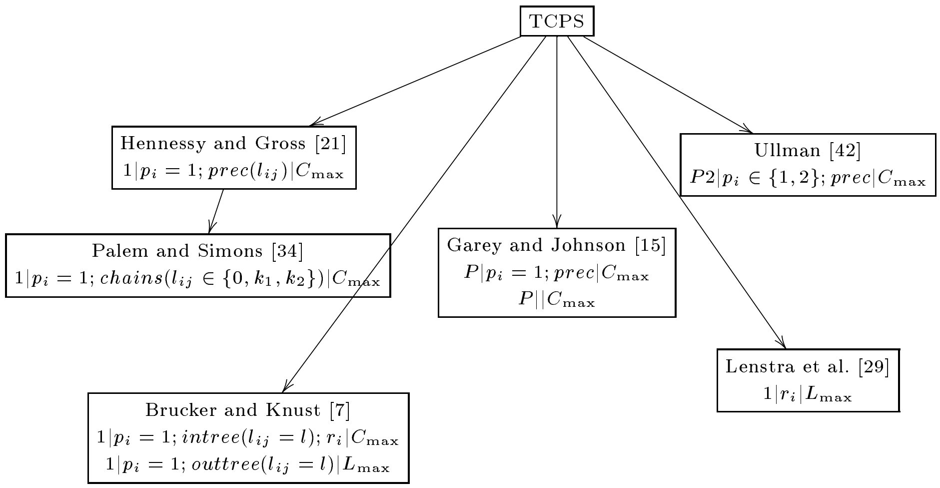

Fig. 6. NP-hardness reductions. All problems listed are known to be NP-hard.

runs in $O(ne + e' \log n)$ where e' is number of edges after transitive closure. The latter algorithm runs in $O(n^2)$, excluding the time to perform transitive reduction.

(3) Moving to the third group, our algorithm can be specialized to the problem of precedence constrained scheduling for two processors in the presence of release-times and deadlines. The algorithm for this case is due to an early result by Garey and Johnson [14], which runs in $O(n^3)$ time.

Ignoring both release-times and deadlines and considering only the makespan minimization case,

(1) We can generalize one of the recent results due to Finta and Liu [10] and show that modified deadlines computed with our algorithm can be used with list-scheduling to minimize the makespan on one processor with arbitrary precedence constraints, individual processing times $p_i \geq 1$ and zero or unit latencies; the work in [10] proves this fact for unit latencies only.

(2) Our algorithm specializes to most of the previously known cases such as the classical two-processor scheduling due to Coffman and Graham [9], again via the unification in [33].

3.2 NP-Hardness

The generic TCPS problem is NP-complete and subsumes many simpler problems which are also known to be NP-complete. Figure 6 summarizes these results.

Hennessy and Gross [21] are the first to show that precedence constrained scheduling with arbitrary latencies on one single pipeline is NP-hard; other proofs are subsequently rediscovered independently in [19; 31; 10]. Palem and Simons [34] strengthen this result to chain precedences and latencies of 0 and two other values. This implies that the problem is NP-hard even if the latency function l_{ij} is invariant under j, i.e. $l_{ij} = l_{ik}$ for all j, k.

Brucker and Knust [7] show that when all latencies are restricted to be a constant, the scheduling problem is NP-hard in the presence of release-times and in-tree precedences. Note that this problem is similar to the polynomially solvable instances consider in [8].

On the multi-processor scheduling front, Garey and Johnson [15] have shown that UET precedence constrained scheduling, and multiprocessor scheduling with variable processing time without precedence constraints, are both NP-hard. Introducing individual processing times into a problem seems to raise its complexity in nearly every case. For example, Ullman [42] has shown that when the processing times are restricted to 1 and 2, the problem is NP-hard on two processors in the presence of precedence constraints. However, further restricting the problem to unit processing time makes it trivially polynomially decidable.

Moving onto the case when all precedence constraints are eliminated. Lenstra et al. [29] show that the one processor scheduling problem with individual processing times and in the presence of release-times and deadlines, is NP-hard. However, when the release-times are also omitted, the problem becomes polynomially solvable with Lawler's algorithm [28]. Similarly, when all processing times are restricted to one, optimal schedules can be computed using Jackson's rule [25]. Note that in both of these polynomially solvable cases, introducing precedence constraints does not complicate the problem, since precedence constraints can be completely encoded by modified deadlines.

3.3 Other Algorithms

Many other algorithms have been developed to add "intelligence" to the basic list-scheduling algorithm. These algorithms are typically developed in more pragmatic settings, and allow a wider range of time-constraints; for example, instead of just latencies, these frameworks may also allow maximal separation constraints, i.e., the maximal time distance two instructions may be separated[6]

Su et al.[40] and Allan et al.[2] introduce a notion of *foresight*, a limited form of lookahead which gauge the the effect of each scheduling step on the remaining unscheduled instructions. Other extensions and improvements [44; 1; 3; 4] have been made to the original idea. These algorithms all keep track of some measure of how flexible the unscheduled instructions are during the list scheduling phase and use these to dynamically reprioritize the set of ready instructions. Theoretical analyses of these techniques are lacking, however, although in practice they are extremely useful.

Thus we view these techniques as orthogonal to our work, although our work here has shown that under certain limited machine models the extra "smarts" in the list scheduler is unnecessary.

4. THE ALGORITHMIC FRAMEWORK

4.1 List-Scheduling

List-scheduling is the framework common to many scheduling algorithms. The basic intuition of why this framework works well in practice is that frequently

[6]This is a variant of the directed bandwidth problem [31; 18].

(i) the complex constraints on instructions can be completely encoded *a priori* as the ranks used in list-scheduling; and *(ii)* optimal schedules are greedy. For example, previous results in [8; 14; 34] are illustrative of this technique. Staying within the list-scheduling framework is appealing and holds promise since it has been successfully validated in pragmatic settings via stable and practically efficient implementations [21; 43].

List-scheduling consists of the following steps.

(1) **Compute ranks** For each $i \in \mathbf{I}$, compute its rank.
(2) **Create priority list** Rank the instructions in non-increasing order of their ranks, with ties broken arbitrarily. Call the resulting priority list of instructions $\mathcal{L}$.
(3) **Greedy list-scheduling** Construct a schedule $\sigma : \mathbf{I} \to \mathbf{N}$ from $\mathcal{L}$, as follows:
   (a) For each successive time-step t, whenever a functional unit remains unassigned, assign to it the first *ready* instruction j on the list $\mathcal{L}$. A ready instruction is an instruction i that satisfies *(i)* $r_j \leq t$, and *(ii)* $\sigma(i)+l_{ij}+p_i \leq t$ for all $i \in pred_j$, Note that if $l_{ij} = -1$, j may be scheduled at the same time-step as i.
   (b) Instructions are removed from $\mathcal{L}$ as they are assigned, and this process repeats until all the instructions have been scheduled.

4.2 Backward Scheduling

We use the *modified deadlines* of instructions as ranks during list-scheduling; i.e. instructions with lower modified deadlines are given higher priority. The modified deadline d'_i of instruction i is an upper-bound on its completion time in any feasible schedule, i.e. in all feasible schedules, i must complete by time d'_i.

Modified deadlines computation repeatedly invokes a *backward scheduling* subroutine BackwardSchedule, which is a generalization of the backward scheduling process described in Palem and Simons [34].

Formally, backward scheduling solves the following *maximization* problem.
INPUT: an instruction i, a set of instructions S distinct from i, release-times r' and deadlines d' on $S \cup \{i\}$.
OUTPUT: the latest completion time $\sigma'(i) + 1$ for i in all schedules σ' on $S \cup \{i\}$ that satisfy the following set of constraints.

(1) **(latency constraints)** $\sigma'(i) + 1 + l^+_{ij} \leq \sigma'(j)$ for all $i \prec j$ and $j \in S$,
(2) **(resource constraints)** for all $t \in \mathbf{N}$, $|\{j \in \{i\} \cup S \mid \sigma'(j) = t\}| \leq m$, and
(3) **(release-times and deadlines)** $r'_j \leq \sigma'(j) < d'_j$ for all $j \in \{i\} \cup S$.

If no feasible schedule σ' exists, return "infeasible."

We call a schedule σ' that satisfies these constraints a *backward schedule* of i. If a backward schedule maximizes the completion time of i, we call this an *optimal* backward schedule. Note that in the backward schedule precedence and latency constraints between instructions from the set S are ignored.

4.3 Computing Modified Deadlines

Intuitively, instruction i must complete by $\sigma'(i) + 1$ in all feasible schedules; thus we can replace the deadline of i with $\sigma'(i) + 1$ without altering the feasibility of the

Instruction	$I1$	$I2$	$I3$	$I4$	$I5$	$I6$	$I7$	$I8$
Release-times r	0	1	0	0	0	0	6	6
Deadlines d	2	2	9	9	7	9	9	8
Initialized release-times r'	0	1	3	1	3	7	6	6
Initial modified deadlines d'	1	2	7	5	7	9	8	8

Fig. 7. Initial modified release-times and deadlines.

original problem. The modified release-times r' and deadlines d' can be computed by iterating the backward scheduling algorithm. A naïve method simply applies the backward scheduling primitive on each instruction i, and update d'_i with the result of

$$\textsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$$

until d' converges. It can be shown that any iteration order that visits all instructions must converge in at most n^2 iterations. (In Section 6 we will show a faster scheme can converge in $O(n)$ iterations.) To be more precise, this algorithm is:

(1) Initialize r' and d' so that they are *consistent* with the precedence and latencies (i.e. $r'_j + l^+_{jk} + 1 \le r'_k$ and $d'_j + l^+_{jk} + 1 \le d'_k$ for all $j, k \in \mathbf{I}$):

$$r'_j := \max\{r_j, \max\{k \in pred_j \mid r_k + l^+_{kj} + 1\}\}$$
$$d'_j := \min\{d_j, \min\{k \in succ_j \mid d_k - l^+_{jk} - 1\}\}$$

For instance, using our example DAG presented in Section 2, the release-times and deadlines are initialized to values listed in Figure 7.

(2) Then repeatedly pick some instruction i from $\mathbf{I}$ in an arbitrary order, and run

$$d'_i := \textsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$$

to compute the new modified deadline of i. Rerun the modified deadlines initialization step

$$d'_j := \min\{d'_j, \min\{k \in succ_j \mid d'_k - l^+_{jk} - 1\}\}$$

for all $j \in \mathbf{I}$ in reverse topological order of the DAG in order to keep the modified deadlines consistent with the latencies. Repeat this step until stability is reached, i.e. until no more changes can be made to any component of d'. If BackwardSchedule returns "infeasible," or for some i, $r'_i \ge d'_i$, then no feasible schedule exists.

For example, a (backward) topological sort for the DAG is

$$I8, I6, I5, I7, I3, I4, I2, I1.$$

In the absence of any other preference, we will visit instructions in this order when invoking the subroutine BackwardSchedule. The first pass of this computation is presented in Figure 8. Observe that in this example the naïve iteration scheme has actually stabilized after the first iteration (with our ordering choice).

Inst.	Backward schedules										d'
	Time	0	1	2	3	4	5	6	7	8	
$I8$		$I1$	$I2$	$I4$	$I3$	$I5$		$I7$	$I8$	$I6$	8
$I6$									$I8$	$I6$	9
$I5$					$I3$		$I5$	$I7$	$I8$	$I6$	6
$I7$		$I1$	$I2$	$I4$	$I3$		$I5$	$I8$	$I7$	$I6$	8
$I3$						$I5$	$I3$	$I7$	$I8$	$I6$	6
$I4$			$I2$	$I4$		$I5$	$I3$	$I7$	$I8$	$I6$	3
$I2$			$I2$	$I4$		$I5$	$I3$	$I7$	$I8$	$I6$	2
$I1$		$I1$	$I2$	$I4$		$I5$	$I3$	$I7$	$I8$	$I6$	1

Fig. 8. Backward schedules and modified deadlines.

4.4 Stability and Consistency

In general, any modified deadline computation scheme that invokes the BackwardSchedule subroutine should have two desirable properties: *stability*[7] and *consistency*, as defined below.

DEFINITION 1. *Let d' be the modified deadlines computed from some iterative scheme $\mathcal{A}$. Then we say that d' is* stabilized *iff*

$$d'_i = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d').$$

We say i is stable *at some step p of algorithm $\mathcal{A}$ iff d^p_i, the modified deadline computed at the pth step for i, is the same as d'_i, where d' is the set of final stabilized modified deadlines.*

DEFINITION 2. *Let r' and d' be the modified release-times and deadlines. Then we say r' and d' are* consistent *iff in all feasible schedule σ, $r_i \le r'_i \le \sigma(i) < d'_i \le d_i$.*

We say that a modified deadline computation scheme $\mathcal{A}$ is *correct* iff it computes stable and consistent modified release-times and deadlines for all feasible instances. Intuitively, stability guarantees that we cannot obtain a tighter estimate for the modified deadlines by reiterating the BackwardSchedule subroutine; while consistency guarantees that whenever a feasible schedule exists, it always satisfies the modified release-times and deadlines, and vice versa. It implies that we can always replace the release-times and deadlines by the modified release-times and deadlines without altering the problem.

LEMMA 1. *Modified deadlines computed by iterating* BackwardSchedule *until the fix-point is reached are stable and consistent.*

PROOF. It is sufficient to see that each invocation of the BackwardSchedule algorithm is solving a relaxed instance of the original scheduling problem. □

5. OPTIMALITY

As noted in Section 3, the general TCPS problem is NP-hard. However, we can still obtain optimal solutions with our framework when restricted to the following types of instances.

[7]This is also called internal consistency by some authors.

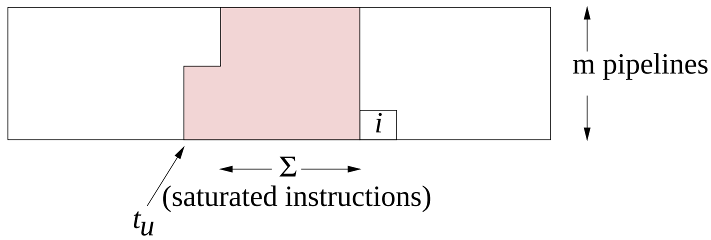

Fig. 9. Schedule in case 1 of Theorem 2.

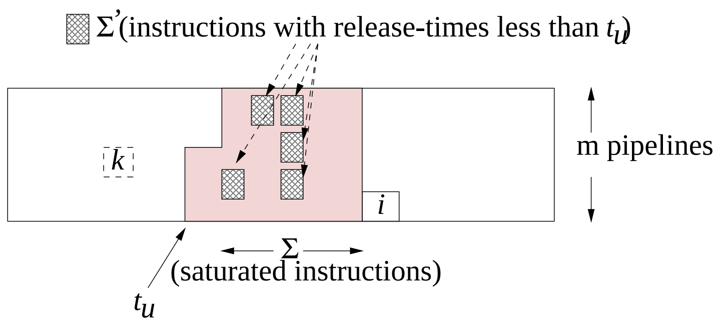

Fig. 10. Schedule in case 2 of Theorem 2.

(I) one pipeline and latencies of 0 or 1, with deadlines and release-times: $(1|p_i = 1; prec(l_{ij} \in \{0,1\}); r_i|L_{\max})$,

(II) m pipelines, a monotone interval-order precedences and latencies, with deadlines and release-times: $(P|p_i = 1; int-order(mono\ l_{ij}); r_i|L_{\max})$,

(III) two pipelines, latencies of -1 or 0 with deadlines and release-times: $(P2|p_i = 1; prec(l_{ij} \in \{-1,0\}); r_i|L_{\max})$,

(IV) m pipelines, constant latencies l, intree precedences, and deadlines: $(P|p_i = 1; intree(l_{ij} = l)|L_{\max})$.

(V) m pipelines, constant latencies l, outtree precedences, and release-times: $(P|p_i = 1; outtree(l_{ij} = l); r_i|C_{\max})$.

For makespan optimization problems, we assume that we are asking a decision question of whether all instructions can complete by some time t.

Note that the solvability of **(I)** directly implies that of **(III)**, given the well-known reductions between a m-processor scheduling problem to a corresponding single machine problem with latencies of $m-1$. Similarly, the solvability of **(IV)** implies that of **(V)**, since these problems are symmetrical.

THEOREM 2. *Using the modified release-times and deadlines computed in our algorithm framework, list-scheduling produces a feasible schedule for instances of type* **(I)**-**(V)**, *iff one exists.*

In the proof below, we will consider all instances concurrently in order to emphasize the similar structures in their arguments. This similarity is unsurprising since almost all modified deadline computations schemes described in the literature, which are subsumed by our algorithm in many cases, can be seen as (further) relaxations of the backward scheduling algorithm used in our framework. For instance, the classical critical path heuristic as be seen as backward scheduling while ignoring all resource constraints.

The optimality proof for case **(IV)** has previously appeared in [8]. It also applies to our algorithm since the proof only depends on consistency of the modified deadlines.

PROOF. Suppose we are given a feasible instance but list-scheduling produces an infeasible schedule σ. Then some instruction must have missed its deadline, and thus its modified deadline. Let instruction i be the first instruction in the schedule that misses its modified deadline, i.e. $\sigma(i) \geq d'_i$. If there are multiple candidates then we may arbitrarily choose one.

We consider an instruction j to be *saturated* if $d'_j \leq d'_i$. We say that a time-step t is saturated if it is scheduled with m saturated instructions. Otherwise, it is unsaturated. Let t_u be the latest unsaturated time-step less than $\sigma(i)$. For convenience, we let t_u be -1 if there are no unsaturated time-steps before $\sigma(i)$. Consider the set of saturated instructions $\Sigma = \{j \in \mathbf{I} \mid t_u < \sigma(j) < \sigma(i)\} \cup \{i\}$. Let $\Sigma' = \{j \in \Sigma \mid r'_j \leq t_u\}$ be the subset in Σ with release times that is less than or equal to t_u.

Case 1. If $|\Sigma'| = 0$ (see Figure 9) then by the pigeon hole principle no feasible schedule exists as any feasible schedule must fit $|\Sigma| = m \times (\sigma(i) - t_u - 1) + 1$ instructions into the time-steps in the range $[t_u + 1, \sigma(i) - 1]$ on m pipelines. This contradicts the assumption that the instance is feasible.

Case 2. We assume $|\Sigma'| \geq 1$ (see Figure 10.) Since t_u is unsaturated and since list-scheduling is greedy[8], we can infer that for all instructions $j \in \Sigma'$, there must exist a *constraining* instruction k that is a predecessor of j such that $\sigma(k) + l^+_{kj} + 1 = \sigma(j)$.

In the following, our analysis will be decomposed into different cases, each corresponding to a different class of instances.

Case 2.1 **(I)**. Since time-step t_u is unsaturated, $m = 1$ and latencies are restricted to $\{0, 1\}$, the instruction k scheduled at time-step $t_u - 1$ must be a predecessor of all instructions in Σ'. From the way the backward scheduling is performed, it must be the case that $\sigma(k) \geq d'_k$. This contradicts the fact that i is the first to miss its modified deadline.

Case 2.2 **(II)**. Recall that in an interval order, given two instructions i and j, either $pred_i \subseteq pred_j$ or $pred_j \subseteq pred_i$. Let j' be an instruction in Σ' such that $|pred_{j'}|$ is minimal. By the property of an interval order, instructions in $pred_{j'}$ are predecessors of all instructions in Σ'. Let k in an instruction in $pred_{j'}$ such that $\sigma(k) + l^+_{kj'} + 1 = \sigma(j')$. By the monotone property, it must be true that $l^+_{kj'} \leq l^+_{kj}$ for all instructions j in Σ'. A feasible schedule can only exist if at least one instruction in Σ' is scheduled at time t_u or earlier. But from the way the backward scheduling is performed, we know that $\sigma(k) \geq d'_k$, which leads to a contradiction.

---

[8]List scheduling will choose a saturated instruction over an unsaturated one whenever possible, since the former has higher priority.

Case 2.3 **(III)**. Since all latencies are restricted to $\{-1, 0\}$ and the number of pipelines are restricted to two, there must be an instruction k scheduled at time t_u such that it is a predecessor of all instructions in Σ'. Since the other slot at time step t_u is either idle or occupied by an unsaturated instruction, by greediness of list-scheduling this implies that for all instructions $j \in \Sigma'$, $l^+_{kj} \geq 0$. However, this means that k must have missed its modified deadline, from the way the backward scheduling is performed. This again contradicts the fact that i is the first instruction to miss its deadline.

Case 2.4 **(IV)**. This case involves intrees precedences, constant latencies of l and deadlines.

Assume $t_u = \sigma(i) - 1$. Then the reason why i is not scheduled at time t_u is because i's immediate predecessor k is scheduled at time $\sigma(i) - l - 1$. But this implies k must have missed its modified deadline, contradicting the choice of i.

Assume $t_u < \sigma(i) - 1$. There must be m saturated instructions scheduled at time step $t_u + 1$. Each of these instructions must have a different immediate predecessor scheduled at time $t_u - l$, since the precedences are in the form of intrees. Repeating the same argument inductively for the instructions at time steps $t \in [t_u + 2, \sigma(i) - 1]$, we can infer that each instruction scheduled at time t has a unique predecessor scheduled at time $t - l - 1$. This implies that i also has a predecessor k scheduled at time $\sigma(i) - 1 - l$. By the way the modified deadline is defined k must have missed its modified deadline, which contradicts the choice of i.

□

6. EFFICIENT IMPLEMENTATION

In this section we will present an efficient implementation of the framework presented in Section 4.

6.1 Backward Scheduling

The backward scheduling problem defined in Section 4.2 can be solved by reducing it into a sequence of simple decision questions. The reduction goes as follows.

We look for the maximal start time of i in the backward schedule in the range $[p, q] = [\max\{d'_i - 2n, r'_i\}, d'_i - 1]$. Lemma 4 presented later shows that an optimal backward schedule can be found within this range, if one exists. We will gradually narrow this range until it cannot be narrowed any further. For each given range $[p, q]$, we set the release-time of instruction i to p and its deadline to $q + 1$. The latency constraints between i and $j \in S$ can be encoded as a new set of release-times for j relative to this provision deadline. The problem then simplifies to a *decision* problem of scheduling unit-time tasks with release-times and deadlines (without precedence and latency constraints):

$$P|p_i = 1; r_i|L_{\max}$$

This well-known problem can be easily solved as follows.

LEMMA 3. *If a feasible schedule for*

$$P|p_i = 1; r_i|L_{\max}$$

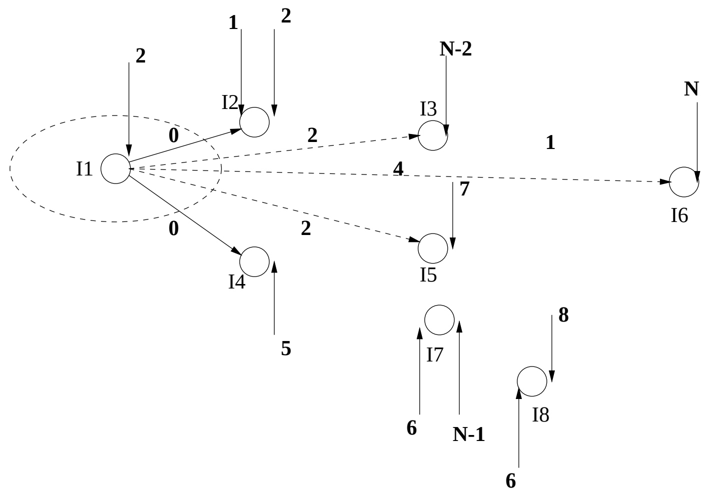

Fig. 11. Backward Scheduling $I1$

exists, then it can be found by list-scheduling with Jackson's rule [25]: schedule a ready instruction with the earliest deadline whenever possible.

For example, suppose we want to compute the optimal backward schedule of $I1$ in Figure 3. The backward scheduling problem asks us to find the maximal completion time of $I1$ in the problem presented in Figure 11. This DAG represents the backward scheduling relaxation of the original problem. Note that dashed edges in this figure represent implied transitive edges from the original graph. For instructions without any original deadline constraints, such as $I6$, we have added an arbitrary and sufficiently large deadline N.

We first try and see if $I1$ can be scheduled at time 1, i.e. complete by time 2, which is equal to its initial modified deadline. By fixing $I1$ at time 1 in the schedule, we can reduce all edges with latencies into the appropriate release-times, giving the simplified instance shown in Figure 12. Note that there is no feasible schedule for this instance, since the release-time and deadline of instruction $I2$ are both equal to 2.

Continuing, we schedule $I1$ at time 0, and obtain the simplified instance shown in Figure 13. This instance is feasible, and thus the optimal backward schedule time of $I1$ in Figure 11 is $\sigma(I1) + 1 = 1$.

6.2 Fast List Scheduling

Since the unit-time tasks list-scheduling decision problem has to be solved repeatedly during the backward scheduling process, it is important to minimize its complexity. The basic intuition is that we can replace a straight forward implementation based on priority queue, which takes $O(n \log n)$ time, by a faster implementation, *if* we are given the instructions already ordered by their release-times and deadlines. This variant of the algorithm can be implemented using the union-find [41] data structure in $O(n\alpha(n))$ time and $O(n)$ space. This algorithm is described below.

Let $\{r_1, \dots, r_R\}$ denote the ordered set of release times in the input and let

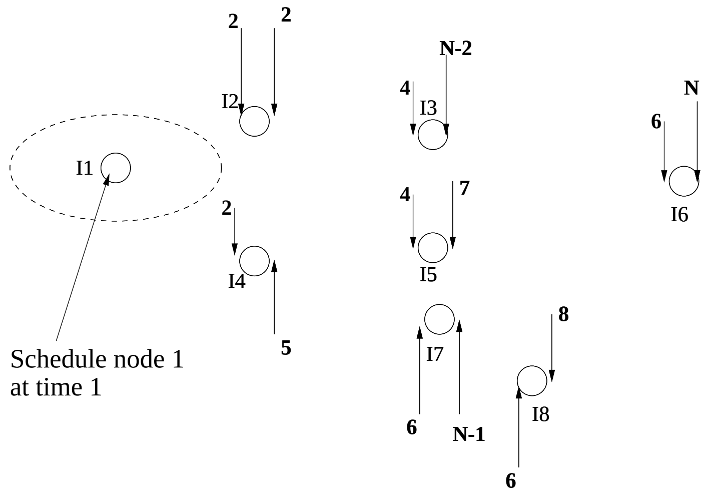

Fig. 12. Backward Scheduling $I1$ ($\sigma(I1) = 1$)

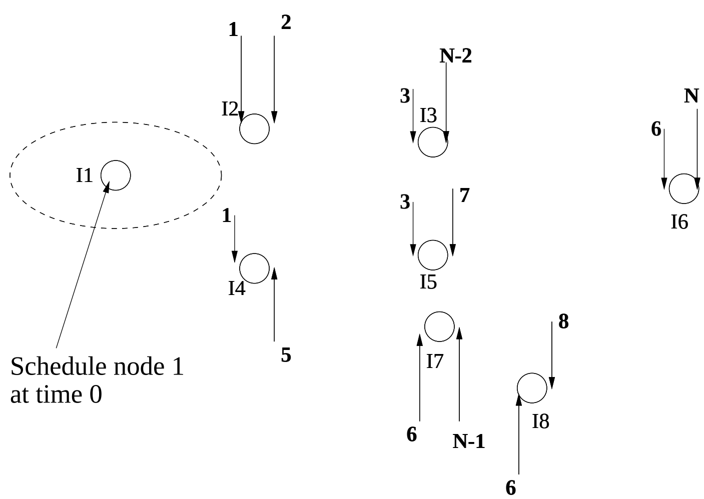

Fig. 13. Backward Scheduling $I1$ ($\sigma(I1) = 0$)

$r_{R+1} = \infty$, i.e. given two ranks p and q, $p < q$ implies $r_p < r_q$. For each rank $\rho \in [1 \ldots R]$, we maintain an interval I_ρ which stands for the time-steps $[r_\rho, r_{\rho+1})$. Each interval contains three data fields *capacity*, *size* and *release_time*. These record the number of available slots in the interval $[r_{\rho+1}, r_\rho)$, the number of slots currently filled with instructions, and the release time r_ρ.

The invariant that we maintain during processing is that no interval is completely full (i.e. $size < capacity$ must be maintained). Whenever an interval becomes full, we enforce the invariant by merging the interval with the next non-full interval. Using the union-find [41] data structure, we can efficiently implement these intervals I_ρ and the merging operation.

Processing proceeds in the following manner. First, we determine $\{r_1, \ldots, r_R\}$ from the release times $\hat{r}$ and the rank for each instruction i, $rank(i)$, such that $\hat{r}_i = r_{rank(i)}$. We process each instruction in non-decreasing order of the deadlines $\hat{d}$. Given an instruction i, we can locate the rank of the instruction $rank(i)$ in constant time. Then $\rho = find(rank(i))$ returns the index of the interval I_ρ that currently contains the first empty slot that i can be feasibly scheduled. The start time of instruction i in the backward schedule is simply the release time r_ρ, plus the number of instructions currently occupied in the interval I_r divided by m. This can be computed using the formula $release\_time + \lfloor size/m \rfloor$. To indicate that the instruction is inserted, we increment the $size$ field of the interval. When the interval I_ρ is completely filled with instructions, we merge the current interval I_ρ with the next adjacent interval. If ρ is the index of the current interval, then the index of the next non-empty interval can be obtained by calling $find(\rho + 1)$. Thus $union(\rho, find(\rho + 1))$ can be used to merge the two intervals. Note that we require *union* to keep the data fields of the non-full interval in the newly merged interval. This bookkeeping can be performed in constant time.

The complete algorithm is listed as below.

ALGORITHM FastListScheduling.
INPUT: a set of instructions S, release times $\hat{r}$, deadlines $\hat{d}$ and two lists L_r and L_d, which contain the instructions in S sorted by release times and deadlines respectively.
OUTPUT: "feasible" iff a feasible schedule σ' exists for S such that $\hat{r}_i \leq \sigma'(i) < \hat{d}_i$.

(1) Rank the release times in the set $\{\hat{r}_j \mid j \in S\}$. Let R denote $|\{\hat{r}_j \mid j \in S\}|$. Assign a rank $rank(j)$ from $\{1 \ldots R\}$ to each instruction $j \in S$, such that $rank(i) < rank(j)$ iff $\hat{r}_i < \hat{r}_j$. Let r_r denote the release time of an instruction of rank r. Let $r_{R+1} = \infty$.

(2) Let *tree* denote an union-find up-tree data structure with 3 addition data fields: *release_time*, *size*, *capacity*. Initialize the data fields of *tree* as follows. For $\rho = 1 \ldots R$, set

$tree[\rho].release\_time := r_\rho$
$tree[\rho].capacity := (r_{\rho+1} - r_\rho) \times m$
$tree[\rho].size := 0$

(3) Process each instruction i in the order of L_d. For each instruction i, execute the following sequence of code to schedule i.

$\rho := find(rank(i))$
$\sigma'(i) := tree[\rho].release\_time + \lfloor tree[\rho].size/m \rfloor;$
$tree[\rho].size := tree[\rho].size + 1;$
if $tree[\rho].size = tree[\rho].capacity$ **then**
  $union(\rho, find(\rho + 1))$
endif

(4) Returns "feasible" if $\sigma'(i) < \hat{d}_i$ for all $i \in S$. Otherwise returns "infeasible."

To see why this algorithm is correct, first observe that each instruction i is never scheduled at a time earlier than its release time $\hat{r}_i$. This is guaranteed by scheduling i in the earliest available time interval $I_{rank(i)}$ or later. Secondly, since instructions are processed in the order of their deadlines, given an instruction i, all time slots between the time-steps $[\hat{r}_i, \sigma'(i) - 1]$ must be filled with instructions with lower or same deadlines as i, with no idle slots in these time-steps. A simple interchange argument as in the proof of Lemma 3 can be used to establish the algorithm's optimality.

6.3 Fast Backward Scheduling

We now present the Algorithm BackwardSchedule, which locates the maximal start time of i in the backward schedule. As stated before, the maximal start time of i can range from $[p, q] = [\max\{d'_i - 2n, r'_i\}, d'_i - 1]$. Our algorithm uses binary search to narrow down on to the range $[p, q]$.

ALGORITHM BackwardSchedule.

(1) We initialize p to $[\max\{d'_i - 2n, r'_i\}, d'_i - 1]$, and q to $d'_i - 1$. In the first phase, we use binary search to locate the maximal q with a feasible backward schedule in the range $[\max\{d'_i - 2n, r'_i\}, d'_i - 1]$. If a feasible q exists, we then move onto the second phase and use binary search to locate the maximal p between the range $[\max\{d'_i - 2n, r'_i\}, q]$. In the first phase, we consider only the instructions in S and initialize S' to S. In the second, we consider both S and i and initialize S' to $S \cup \{i\}$.

(2) For each step of the search, we convert the latencies between i and its successors into release-times as follows. For each $j \in S'$, compute a new set of release-times $\hat{r}$ and deadlines $\hat{d}$ using the following reduction:

$$\hat{r}_j := \begin{cases} \max\{r'_j, q + 1 + l^+_{ij}\}, & \text{if } j \in S \\ p, & \text{if } j = i \end{cases}$$
$$\hat{d}_j := \begin{cases} d'_j, & \text{if } j \in S \\ q + 1, & \text{if } j = i \end{cases}$$

We will call this reduction $\mathcal{R}$.

(3) If this is the first iteration of the search, we construct the lists L_r and L_d as required by FastListScheduling by sorting on the keys $\hat{r}$ and $\hat{d}$. In subsequent iterations, we re-construct the lists L_r and L_d from the old lists computed in the previous iteration in time $O(n)$ as follows.
Scan the old list L_r and partition it into three lists L_1, L_2, L_3. The first list L_1 contains all the instructions j whose release-times $\hat{r}_j$ are unchanged from the

previous iteration. The remaining instructions are partitioned into L_2, and L_3. L_2 contains all instructions j where $\hat{r}_j = r'_j > q + 1 + l^+_{ij}$, while L_3 contains all instructions j where $r'_j \leq q + 1 + l^+_{ij} = \hat{r}_j$.

Consider the task of keeping L_1, L_2 and L_3 sorted in the order of the release-times $\hat{r}$. By definition, L_1 is already in order. Instructions in L_2 must be sorted in the order of r'_j, while instructions in L_2 must be sorted in the order of $q + 1 + l^+_{ij}$, but this is the same as the order of l^+_{ij}. Since r'_j and l^+_{ij} do not change between iterations we can precompute these orderings once before the binary search begins, and re-sort L_2 and L_3 in linear time. The new list L_r can be computed by merging L_1, L_2 and L_3.

The new list L_d can be reconstructed from the old L_d by removing the instruction i from the old list and reinserting it into the correct sorted position.

(4) Run FastListScheduling on the inputs S', $\hat{r}$, $\hat{d}$, L_r, and L_d to compute a backward schedule σ'.

(5) Repeat from (2) until the binary search finishes. If σ' is feasible, return $p + 1$ as the modified deadline of i.

(6) Otherwise, return "infeasible."

6.4 Fast Modified Deadline Computation

The naïve iterative scheme for computing modified deadlines presented in Section 4 does not guarantee quick convergence in general, since each instruction may have to be visited multiple times before the modified deadlines converge to a fixed value. In our previous example we have been fortunate in our choice of visitation ordering such that only two iterations are sufficient[9]. The crux of reaching a correct set of modified deadlines quickly is to choose the order in which instructions are processed carefully.

It turns out that a simple ordering of instructions exists which guarantees that the modified deadlines converge in one single pass, which is optimal. This scheme is implemented in the Algorithm ModifiedDeadlines, shown below.

ALGORITHM ModifiedDeadlines.
INPUT: An instance of the UET TCPS problem.
OUTPUT: A set of modified release-times r' and deadlines d'; or "infeasible" if the instance is infeasible.

(1) Initialize the modified release-times r' and deadlines d' in topological order[10] such that they are consistent with the partial order $\prec$ and the latencies: for all $j \in \mathbf{I}$,

$$r'_j := \max\{r_j, \max\{k \in pred_j \mid r_k + l^+_{kj} + 1\}\}$$
$$d'_j := \min\{d_j, \min\{k \in succ_j \mid d_k - l^+_{jk} - 1\}\}$$

(2) Set $P := \emptyset$. Set $U := \mathbf{I}$.

[9] One pass is needed to converge and another is needed to check for convergence.
[10] Update deadlines in reverse topological order, starting with instructions without successors.

Inst.	Backward schedules										d'
	t	0	1	2	3	4	5	6	7	8	
$I6$										$I6$	9
$I7$									$I7$	$I6$	8
$I8$								$I7$	$I8$	$I6$	8
$I5$							$I5$	$I7$	$I8$	$I6$	6
$I3$					$I5$		$I3$	$I7$	$I8$	$I6$	6
$I4$				$I4$		$I3$	$I5$	$I7$	$I8$	$I6$	3
$I2$			$I2$	$I4$		$I3$	$I5$	$I7$	$I8$	$I6$	2
$I1$		$I1$	$I2$	$I4$		$I3$	$I5$	$I7$	$I8$	$I6$	1

Fig. 14. Backward schedules and modified deadlines computed with Algorithm 3

(3) Pick an $i \in U$ such that r'_i is maximal in U and such that all successors of i have been processed; break ties arbitrarily. Compute the new modified deadline of i with $d'_i := \mathsf{BackwardSchedule}(i, P, r', d')$.

(4) Rerun $d'_j := \min\{d'_j, \min\{k \in succ_j \mid d'_k - l^+_{jk} - 1\}\}$ for all $j \in U - \{i\}$ in reverse topological order.

(5) Remove i from the set U. Add i to the set P. Repeat from Step (3) if U is non-empty.

(6) If any invocation of BackwardSchedule returns "infeasible", then return "infeasible" as result. Otherwise, return r' and d' as the modified release-times and deadlines respectively.

Note that there is one key difference between this algorithm and the naïve iteration scheme: this algorithm invokes BackwardSchedule with only the previously processed elements as parameters.

6.5 An Example

Let us now test Algorithm ModifiedDeadlines on our running example. The release-times r' and deadlines d' are initialized to

$$(0, 1, 3, 1, 3, 7, 6, 6)$$

and

$$(1, 2, 7, 5, 7, 9, 8, 8)$$

as in Step (1) of the algorithm. We have $r'_6 \geq r'_7 \geq r'_8 \geq r'_5 \geq r'_3 \geq r'_4 \geq r'_2 \geq r'_1$. We process the instructions in the non-increasing order of r'_i, which is $I6, I7, I8, I5, I3, I4, I2$ and $I1$. Figure 14 displays the backward schedules and the new modified deadlines computed in each invocation of BackwardSchedule. Note that instruction $I5$ must be scheduled at time 5 and no later in its backward schedule since the instructions $I6, I7$, and $I8$ cannot start earlier than time 6.

By list-scheduling the instructions in non-decreasing order of their modified deadlines, we can obtain the optimal schedule in Figure 4.

6.6 Correctness

We will now show that the algorithms BackwardSchedule and ModifiedDeadlines are correct.

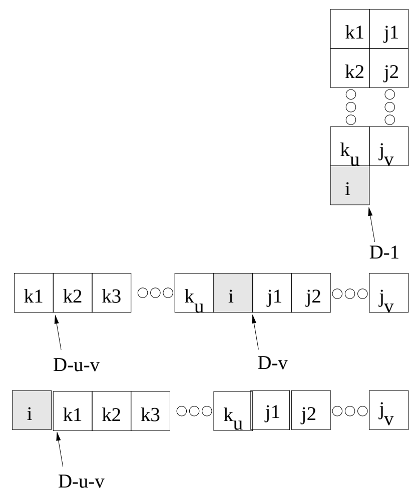

Fig. 15. Effect of Lemma 4

We begin with the following technical lemma, which shows that it is sufficient to invoke FastListScheduling at most $O(\log n)$ times in each invocation of BackwardSchedule.

LEMMA 4. *Algorithm* BackwardSchedule *needs to invoke* FastListScheduling *at most $O(\log n)$ times to find a feasible backward schedule, or no feasible schedule exists.*

PROOF. Recall that the modified deadlines passed to routine BackwardSchedule are initialized in Steps (1) and (4) of ModifiedDeadlines so that they are kept consistent with the latencies, i.e.

$$d'_i \leq \min\{j \in succ_i \mid d'_j - l^+_{ij} - 1\}$$

Let d^0_i denote the modified deadline of i initialized in this manner. In the absence of release-time constraints and resource constraints (e.g. $m \geq n$), Algorithm BackwardSchedule will not compute a tighter estimate for d'_i.

With the presence of only resource constraints (i.e. ignoring release-time constraints) the modified deadline computed by BackwardSchedule is at most $\lceil |succ_i|/m \rceil$ time-steps less than d^0_i. This can be seen by converting a feasible backward schedule computed for a machine with infinite number of pipelines to a feasible backward schedule with only m pipelines, by shifting instructions forward.

Now consider incorporating release-time constraints. Adding release-time constraints to some instruction j can only delay j in the backward schedule. This may in turn cause a sequence of instructions to be shifted forward, decreasing the maximal feasible completion time of i in σ'. Since there are at most $n-1$ instructions in the set $succ_i \cup indep_i$, i can be moved by at most an additional $n-1$ time-steps.

Thus starting from the initial estimate d^0_i, the final modified deadline of d'_i must be at least $d^0_i - 2n$, assuming that a feasible (backward) schedule exists. This implies that the binary range search will terminate in $O(\log n)$ iterations. □

Figure 15 illustrates the some of the effects described in the above lemma. Consider a backward schedule for i on one pipeline where $succ_i = \{j_1, \ldots, j_v\}$ and $indep_i = \{k_1, \ldots, k_u\}$. Let all latencies be zero and D be the deadlines of all instructions. In addition, let $r'_{k_x} = D - u - v$ for all $x = 1 \ldots u$. The top backward schedule is constructed assuming infinite resources are available and by ignoring all release-times. When resource constraints are added, i has to be moved forward $v - 1$ time-steps, as in the backward schedule in the middle of the figure. Finally, when release-times are also incorporated into the backward schedule, instruction i has to be shifted forward by an additional u time-steps, as in the bottom of the figure.

Suppose the algorithm BackwardSchedule completes successfully and at the end the binary search ends in the range $[p, q]$. It is easy to see that the reduction $\mathcal{R}$ used in the algorithm preserves the latency, release-times and deadlines constraints. Thus the backward schedule computed must be feasible. From the way q is computed, i cannot start after q in any feasible backward schedule, i.e. q is an upper-bound on the maximal start time of i. Thus it is clear that if there is a backward schedule, the algorithm will return a feasible backward schedule.

If $p = q$, the correctness of the algorithm holds trivially. Now suppose $p < q$. We have to show that i is scheduled as late as possible, i.e. there does not exist another feasible backward schedule associated with the range $[p', q']$, where $p' > p$ and $q' < q$. The following lemma shows this using an inductive argument.

LEMMA 5. *Algorithm* BackwardSchedule *computes the maximal completion time of i in all feasible backward schedules of i.*

PROOF. Let σ' denote the final backward schedule computed in Algorithm BackwardSchedule, and let $[p, q]$ denote the range associated with σ'. Notice that the instruction i must be scheduled at time p in σ', by the fact that p is maximal.

Consider the structure of σ'. Without loss of generality we may assume $p < q$. Let the sets $\Pi_0 \subseteq \Pi_1 \subseteq \Pi_2 \subseteq \ldots$ denote instructions scheduled in the ranges $[p+1, s_0], [p+1, s_1], [p+1, s_2]$, etc. We will show that in each range the following properties hold:

(1) there is no idle slot,

(2) for any instruction $j \in indep_i$ scheduled in the range, $r'_j \geq p + 1$.

The time-steps $s_0 \leq s_1 \leq s_2 \leq \ldots$ are defined inductively as follows:

(1) **Base case:** let $s_0 = q$. By the definition of p, all successors of i with latency of 0 or greater must be scheduled after q in any backward schedule. Thus all instructions in Π_0 must be from the set $indep_i \cup \{j \in succ_i \mid l^+_{ij} = -1\}$. By the maximality of p, clearly there cannot be any idle slots in the interval $[p+1, s_0]$. Furthermore, for all $j \in \Pi_0 \cap indep_i$, it must the case that $r'_j \geq p+1$. Otherwise, we may exchange the positions of i and j and obtain another feasible backward schedule where i is scheduled later than p, which would violate the maximality of p.

(2) **Inductive case:** Define

$$s_{n+1} \stackrel{\text{def}}{=} \max\{d'_j - 1 \mid j \in \Pi_n\}$$

Notice that there cannot be any idle slot between $[s_n + 1, s_{n+1}]$. Otherwise, by the definition of s_{n+1} we can find some instruction $j \in \Pi_n$ with deadline of $d'_j = s_{n+1} + 1$ and move it to this idle slot. This preserves the feasibility of the schedule and creates an empty slot in the range $[p+1, s_n]$. By the induction hypothesis, we can then move i into some slot in $[p+1, s_0]$ via a sequence of exchanges. Similarly, notice that all instructions $j \in (\Pi_{n+1} - \Pi_n) \cap indep_i$ must have release-times at least $p+1$. Otherwise, we may move i into some slot in $[p+1, s_0]$ via a sequence of exchanges as in the previous case, and move j to time-step p, where i is scheduled. This implies that there are no idle slots in the range $[p+1, s_{n+1}]$ and $j \in \Pi_{n+1} \cap indep_i$ implies $r'_j \geq p+1$.

Let $s = \max_{j \in 0 \ldots \infty} s_j$, and $\Pi = \cup_{j \in 0 \ldots \infty} \Pi_j$. Then:

(1) There are no idle slots in the time-steps $[p+1, s]$, and
(2) All instructions in $\Pi \cap indep_i$ must have release-times at least $p+1$ and deadlines at most $s+1$.

Consider the effect of replacing $[p, q]$ by some $[p', q']$ where $p < p'$ and $q' < q$, on the reduction $\mathcal{R}$. Changing q to q' only affects the release-times $\hat{r}$ of the successors of i, and the deadline $\hat{d}_i$. Changing p' to p to increases the release-time $\hat{r}_i$. Since $p < p' \leq q' < q \leq s$, the new release-times of the instructions in $(\Pi \cap indep_i) \cup \{i\}$ must be at least $p+1$, and the new deadlines must be at most $s+1$. Clearly, all successors of i cannot be scheduled earlier than i in the backward schedule. By the pigeon hole principle no feasible backward schedule can exist in this configuration since we have to pack $m \times (s-p) + 1$ instructions in $m \times (s-p)$ time slots. Thus σ' is indeed optimal and $p+1$ is the maximal completion time of i in all backward schedules. $\square$

Each backward schedule has an interesting subset of instructions that determines its layout. This subset is defined below.

Definition 3. *Let i be any instruction. Let r' and d' denote any modified release-time and deadline assignments. Let*

$$t = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d').$$

then the supporting instructions *for i is the set*

$$\mathsf{Sup}(i, r', t) \stackrel{\text{def}}{=} succ_i \cup \{j \in indep_i \mid r'_j \geq t\}$$

The next lemma shows that only the supporting instructions are relevant during the backward scheduling computation.

Lemma 6. *Suppose i has a backward schedule. If*

$$t = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$$

and

$$t' = \mathsf{BackwardSchedule}(i, \mathsf{Sup}(i, r', t), r', d')$$

then $t = t'$.

Proof. *(1)* Clearly $t \leq t'$. In the proof for Lemma 5 we show that given an optimal backward schedule σ', there must exists some (possibly empty) range $[\sigma'(i) + 1, s]$, where:

(1) i cannot be scheduled after s in any feasible backward schedule, and

(2) there are $m \times (s - \sigma'(i))$ *constraining* instructions scheduled in the range $[\sigma'(i)+1, s]$ and all of them must be scheduled within this range in all feasible backward schedules.

Since all these constraining instructions have release-times at least $\sigma'(i) + 1 = t$, we cannot obtain a later start-time for i in the backward schedule consist of only instructions from the set $\mathsf{Sup}(i, r', t) \cup \{i\}$. Thus $t = t'$ as stated. $\square$

The consequence of this result is that we can compute the modified deadline for an instruction i by considering only its supporting instructions at each step. However, the definition of supporting instructions is somewhat "circular" in nature: we have to know the result of

$$t = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$$

before we can determine the set $\mathsf{Sup}(i, r', t)$. Fortunately, it is sufficient to consider a superset of $\mathsf{Sup}(i, r', t)$, during each backward scheduling step. This is exactly what is done in Algorithm ModifiedDeadlines. Thus we obtain the following theorem.

THEOREM 7. *Algorithm* ModifiedDeadlines *computes consistent and stable modified release-times and deadlines, i.e. for all instruction i,*

$$d'_i = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$$

or else the instance is infeasible.

PROOF. **Consistency.** Trivial from the fact that each invocation of BackwardSchedule is solving a relaxed instance of the original problem.

Stability. We assume that the instance is feasible. We will prove by induction that only stable instructions are removed from the set U and added to P during Step (3) of the algorithm.

Base and Induction case:. Let i be the current instruction chosen to be processed. Let $t = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d')$, where d' denote the current modified deadlines. By Lemma 6 we get

$$t = \mathsf{BackwardSchedule}(i, \mathsf{Sup}(i, r', t), r', d')$$

Consider all the instructions remaining in U. By the order of processing we know that for all $j \in U$, j is not a successor of i and $r'_j \leq r'_i$. Thus $r'_j \leq r'_i < t \leq d'_i$. Consider any deadlines d'' where $r'_k + 1 \leq d''_k \leq d'_k$ for all $k \in \mathbf{I}$. Suppose $t' = \mathsf{BackwardSchedule}(i, succ_i \cup indep_i, r', d'')$, By definition, $\mathsf{Sup}(i, r', t') = succ_i \cup \{k \in indep_i \mid r'_k \geq t'\}$. Since $r'_j \leq r'_i < t' \leq t$, j cannot be a support instructions of i, for any values for d''.

In the base case, i must be a sink instruction without any successors. In the induction case, all processed instructions P are stable by the induction hypothesis. From the processing order we know that $\mathsf{Sup}(i, r', t) \subseteq P$. This implies i is also stable after this step.

By induction on $|P|$, all instructions in P are stable when the algorithm completes. $\square$

6.7 Algorithmic Complexity

In the following worst case complexity analysis, we will assume that the input DAG is not given in transitively closed form. We will instead factor in the time to perform transitive closure. As before, n denotes the number of instructions (nodes) in the input DAG, and e denotes the number of edges.

THEOREM 8. BackwardSchedule *runs in* $O(n \log n \alpha(n) + e)$ *time, where* $\alpha(n)$ *is the functional inverse of the Ackermann function.*

PROOF. By Lemma 4 it is sufficient to invoke FastListScheduling $O(\log n)$ times in BackwardSchedule. Each invocation of FastListScheduling runs in time $O(n\alpha(n))$ since its complexity is dominated by the $O(n)$ union-find operations.

The release-times and deadlines $\hat{r}$ and $\hat{d}$ can be computed in time $O(n)$ in each iteration. The initial lists L_r and L_d are constructed by sorting once during initialization, which takes a total $O(n \log n)$ time. In subsequent iterations, the lists L_r and L_d can be reconstructed from the old lists computed in the previous iteration. This can be done in $O(n)$ time using the method as described in Step (3) of the algorithm.

Given an instruction i, its successors $succ_i$ and the transitive latencies l^+_{ij} can be computed in time $O(n+e)$. Altogether, the total running time of BackwardSchedule is

$$O(\log n)O(n\alpha(n)) + O(n \log n) + O(n + e)) = O(n \log n\alpha(n) + e)$$

□

THEOREM 9. ModifiedDeadlines *runs in* $O(n^2 \log n\alpha(n) + ne)$ *time.*

PROOF. Step (1) of ModifiedDeadlines can be completed in time $O(n^2)$. It takes $O(n \log n)$ time to sort the modified release-times. Since ModifiedDeadlines invokes BackwardSchedule n times and each invocation takes time $O(n \log n\alpha(n) + e)$, total running time is dominated by the time spent in BackwardSchedule, which is $O(n^2 \log n\alpha(n) + ne)$. □

The complexity of our algorithm can be further improved to $O((m + n)n \log n + ne)$, where m is the number of pipelines, using Frederickson's [11] algorithm for scheduling unit-time tasks with integer release-times and deadlines, which runs in $O(n)$ time by applying the off-line union-find algorithm of Gabow and Tarjan [12]. Since m is typically much smaller than n, this is an improvement. In fact, the maintenance of the sorted lists L_r and L_d in ModifiedDeadlines becomes unnecessary when this list-scheduling algorithm is used as the subroutine. Unfortunately, this implementation of the list-scheduling subroutine requires $O(m+n+\delta)$ space, where δ is the difference between the largest deadline and the smallest release-time of an instance. Since this difference is unrelated to n, the number of instructions, and may be arbitrarily larger than n, this change may require super-polynomial space in general.

Finally, minimal tardiness can be computed by running $O(\log n)$ iterations of the above algorithm, using binary search.

7. INSTRUCTIONS WITH NON-UNIT PROCESSING TIMES

In this section we will relax the restriction that all processing times are unit length. We restrict our attention to only single processor problems, since precedence constrained multiprocessor scheduling is NP-complete, even when restricted to two processors and processing times of 1 and 2 [42].

7.1 Minimizing Makespan

It turns out that our algorithm be also used to construct schedules with optimal makespan for a single processor, when instructions have individual processing times of one or more, and in the presence of precedence and latencies of 0 and 1 ($1 \mid prec(l_{ij} \in \{0,1\}), p_j \geq 1 \mid C_{\max}$).

A similar problem with unit-latencies, $1 \mid prec(l_{ij} = 1), p_j \in \mathbf{N}_+ \mid C_{\max}$, has been addressed in [10], using the Coffman/Graham [9] lexicographical rank function for list-scheduling priorities. The optimality proof makes use of a certain rigid structure of list-schedules generated by this rank function, and is quite involved.

In contrast, our scheme for optimal makespan scheduling is very simple: ignore the processing times and treat an instance with individual processing times as a UET instance, and schedule the instructions of the former in the same list-schedule order as in the latter. The basic idea is that the UET schedule and the schedule with individual processing times must have the same number of idle cycles, since latencies are at most one while processing times are at least one; and optimality of one implies the other.

Let $\mathcal{I}$ to denote an instance of $1 \mid prec(l_{ij} \in \{0,1\}), p_j \geq 1 \mid C_{\max}$. Then $\mathcal{I}_u$ denote the same instance with only unit-processing time. The following facts can be observed.

LEMMA 10. *Suppose the instructions occur in the order of $O = i_1, i_2, \ldots i_n$ in a feasible schedule for $\mathcal{I}$, then the same order produces a feasible schedule for $\mathcal{I}_u$, and vice versa.*

PROOF. Trivial. □

THEOREM 11. *Let σ be an optimal schedule σ for an instance of*

$$1 \mid prec(l_{ij} \in \{0,1\}), p_i = 1 \mid C_{\max}$$

then scheduling the instructions in the same order as in σ produces an optimal schedule for

$$1 \mid prec(l_{ij} \in \{0,1\}), p_i \geq 1 \mid C_{\max}$$

PROOF. Clearly a schedule in the order of O is feasible for $\mathcal{I}$. Consider the optimal greedy schedule σ_u for $\mathcal{I}$ and the greedy schedule σ obtained from the order O. Since the processing times are at least one, if σ_u has no idle time steps then σ also has no idle time steps and is trivially optimal.

Now suppose at least one idle time step exists in σ_u. Consider an idle time step t_i in σ_u. By greediness and the fact that the latencies are at most 1, we can infer that the instruction k_i scheduled at time $t_i - 1$ is a predecessor of all the instructions (with latency 1) scheduled after t_i. This implies that an idle time step must also appear after the same instruction k_i in the schedule σ. Since the processing times for all the instructions are at most one, additional idle time steps cannot be created when we convert σ_u into σ, and both schedules σ_u and σ have the same number of idle time steps.

Suppose an optimal schedule σ_{OPT} exists for $\mathcal{I}$ such that σ_{OPT} has fewer number of idle time steps than in σ. Without loss of generality we can assume that σ_{OPT} is a greedy schedule. By Lemma 10 we can find a greedy feasible schedule σ'_u for $\mathcal{I}_u$ by using the same scheduling order as in σ_{OPT}. Since the latencies are at most 1 and the processing times are at least 1, if two instructions i and j are schedule adjacently without idle time in σ_{OPT} the same instructions can be scheduled in σ'_u adjacently without idle time. Similarly, if instructions i, j and k are scheduled consecutively in σ_{OPT} without idle time in between, then the same instructions can be scheduled consecutively in σ_u. But these observations imply that σ'_u must have the same number of idle steps as in σ_{OPT}, which is fewer than the number of idle time steps than σ_u, a contradiction. $\square$

Note that this result depends only on the existence of an optimal scheduling algorithm for the unit processing time problem. Thus a faster algorithm exists. The labelling function by Bernstein and Gertner [5] can be computed in $O(n^2)$ time and is optimal for this problem[11].

A final remark is that the previous result does not depend on the fact that the processing times are integral, and thus any arbitrary rational processing times at least one will suffice.

8. SUMMARY

In this work we investigate the problem of scheduling instructions on RISC and VLIW machines with multiple pipelines in the presence of precedence constraints, latencies, release-times and deadlines, using techniques from deterministic scheduling. While the general scheduling problem is NP-complete, we have developed an algorithm that can optimally solve many subcases. Our algorithm uses the technique of backward scheduling, which is a generalization of many deadline modification techniques known in the literature.

Because of this generality, we believe our algorithm is a good heuristic in practice for scheduling code with timing constraints. Heterogenous pipelines or complex resource constraints are not modelled in our work. However it can be extended in the obvious manner in the list-scheduling steps (including those within the backward scheduling algorithm), by assigning instructions only to those pipelines of the correct functional unit types, but the resulting algorithm would be a heuristic, with no optimality guarantees.

A few interesting questions related to this work remain. One question is: can we extend this "program" of algorithmic unification to other more general scheduling models. For example, can backward scheduling be strengthened to solve related problems that do not have optimal greedy solutions, such as those involving equal but non-unit processing times [39]? We conjecture that this is so, but currently we do not have any definite answers. Progress in this area will be useful in practice, and may also clarify the structure of polynomially solvable scheduling problems.

[11] This algorithm assumes that the input is presented in transitive-closed form, so the overall complexity is that of transitive reduction.

The dual question is: can our algorithm and framework be specialized to obtain faster running times for specific problems? The answer to this question appears to be affirmative. Wu et al. [45] have shown that in some specific cases our backward scheduling relaxation can be loosen in ways that the computation of transitive latencies can be replaced by computation of successors. They use this observation to obtain faster solutions for the 2-processor and the 0/1-latency scheduling problems.

REFERENCES

1. Allan, V., Beaty, S. J., Su, B., and Sweany, P. H. Building a retargetable local instruction scheduler. *Software: Practice and Experience* (Spring 1998).
2. Allan, V., Su, B., Wijaya, P., and Wang, J. Foresighted compaction under timing constraints. *IEEE Transactions on Computers 41*, 9 (1992), 1169–1172.
3. Beaty, S. J. Lookahead scheduling. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)* (Portland, Oregon, Dec. 1992).
4. Beaty, S. J. List scheduling: Alone, with foresight, and with lookahead. In *In Conference on Massively Parallel Computing Systems (MPCS): the Challenges of General-Purpose and Special-Purpose Computing* (May 1994).
5. Bernstein, D., and Gertner, I. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems 11*, 1 (1989), 57–66.
6. Bernstein, D., and Rodeh, M. Global instruction scheduling for superscalar machines. *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991).
7. Brucker, P., and Knust, S. Complexity results for single-machine problems with positive finish-start time-lags. *Osnabruecker Schriften zur Mathematik, Reihe P, Nr. 202* (1998).
8. Bruno, J., Jones, J., and So, K. Deterministic scheduling with pipelined processors. *IEEE Transactions of Computers C-29* (Apr. 1980), 308–316.
9. Coffman, E. G., and Graham, R. L. Optimal scheduling for two-processor systems. *Acta Informatica 1* (1972), 200–213.
10. Finta, L., and Liu, Z. Single machine scheduling subject to precedence delays. *Discrete Applied Mathematics 70* (1996), 247–266.
11. Frederickson, G. N. Scheduling unit-time tasks with integer release times and deadlines. Tech. Rep. CS-81-27, Dept. of Computer Science, Penn. State University, 1982.
12. Gabow, H. N., and Tarjan, R. E. A linear-time algorithm for a special case of disjoint set union. *Proc. of ACM Symposium on Theory of Computation* (1983), 246–251.
13. Garey, M., and Johnson, D. Scheduling tasks with nonuniform deadlines on two processors. *Journal of the ACM 23*, 3 (July 1976), 461–467.
14. Garey, M., and Johnson, D. Two-processor scheduling with start-times and deadlines. *SIAM Journal of Computing 6*, 3 (Sept. 1977), 416–426.
15. Garey, M., and Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.
16. Gibbons, P., and Muchnick, S. Efficient instruction scheduling for pipelined architecture. *Proceedings of the ACM Symposium on Compiler Construction* (1986), 11–16.
17. Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. H. G. R. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics 5* (1979), 287–326.
18. Han, C., and Lin, K. Job scheduling with temporal distance constraints. Tech. Rep. TR-UIUCDCS-R-89-1560, University of Illinois, 1989.
19. Han, C., and Lin, K. Scheduling real-time computations with separation constraints. *Information Processing Letters 42* (1992), 61–66.
20. Hank, R. E., Hwu, W., and Rau, B. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th International Workshop on Microprogramming and Microarchitecture* (1995), pp. 158–168.

21. Hennessy, J., and Gross, T. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems 5(3)* (1983).

22. Hennessy, J., Jouppi, N., Gill, J., Baskett, F., Strong, A., Gross, T., Rowen, C., and Leonard, J. The MIPS machine. *Proceedings IEEE Compcon* (February 1982), 2–7.

23. Hu, T. C. Parallel sequencing and assembly line problems. *Operations Research 9* (1961), 841–848.

24. Hwu, W.-M. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., T. Kiyohara, G. E. H., Holm, J. G., and Lavery, D. M. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing 7* (1993), 229–248.

25. Jackson, J. R. Scheduling a production line to minimize maximum tardiness. Tech. Rep. 43, Management Science Research Project, University of California, 1955.

26. Katevenis, M. *Reduced Instruction Set Computer Architecture for VLSI.* MIT Press, Cambridge, Mass., 1984.

27. Lam, M. Software pipelining: An effective scheduling technique for VLIW machines. *Proceedings SIGPLAN'88 Symposium on Programming Language Design and Implementation* (1988), 318–328.

28. Lawler, E. L. Optimal sequencing of a single machine subject to precedence constraints. *Management Sci.*, 19 (1973), 544–546.

29. Lenstra, J. K., Kan, A. H. G. R., and Brucker, P. Complexity of machine scheduling problems. *Annals of Discrete Mathematics 1* (1977), 343–362.

30. Leung, A., Palem, K., and Pnueli, A. A fast algorithm for scheduling time-constrained instructions on processor with ILP. *The International Conference on Parallel Architectures and Compilation Techniques (PACT '98)* (1998).

31. Leung, J. Y.-T., Vornberger, O., and Withoff, J. D. On some variants of the bandwidth minimization problem. *SIAM Journal of Computing 13*, 3 (1984).

32. Moon, S., and Ebcioglu, K. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings IEEE MICRO-25* (1992).

33. Palem, K. On the complexity of precedence constrained scheduling. Tech. Rep. TR-86-11, University of Texas, Austin, TX, 1986.

34. Palem, K., and Simons, B. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems 5(3)* (1993).

35. Papadimitriou, C., and Yannakakis, M. Scheduling interval-ordered tasks. *SIAM Journal of Computing 8* (1979), 405–409.

36. Radin, G. The 801 minicomputer. *IBM Journal of Research and Development 27*, 3 (1983), 237–246.

37. Rau, B. Iterative modulo scheduling: An algorithm for software pipelining loops. *Proceedings of the 27th Annual Symposium on Microarchitecture* (December 1994).

38. Rau, B. R., and Glaser, C. D. Scheduling scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture* (Oct. 1981), 183–198.

39. Simons, B. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal of Computing 12* (1983), 294–299.

40. Su, B., Ding, S., and Xia, J. Microcode compaction with timing constraints. In *In Proceedings of the 20th Microprogramming Workshop (MICRO-20)* (Colorado Springs, Colorado, Dec. 1987).

41. Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22* (April 1975), 215–225.

42. Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Sciences 10* (1975), 384–393.

43. Warren, H. Instruction scheduling for the IBM RISC system/6k processors. *IBM Journal of Research and Development* (1990), 85–92.

44. Wijaya, P., and Allan, V. H. Incremental foresighted local compaction. In *Proceedings of the 22nd annual international workshop on Microprogramming and microarchitecture* (Dublin, Ireland, Aug. 1989).

45. Wu, H., Yap, R., and Jaffar, J. Private communication, 1999.

Received 10/11/99, Accepted 10/29/00