A fast algorithm for scheduling time-constrained instructions on processors with ILP*

Allen Leung
Courant Institute of Mathematical Sciences
251 Mercer St., New York, NY 10012

Krishna V. Palem
Courant Institute of Mathematical Sciences
251 Mercer St., New York, NY 10012

Amir Pnueli
Weizmann Institute of Science
76100 Rehovot, Israel

Abstract

Instruction scheduling is central to achieving performance in modern processors with instruction level parallelism (ILP). *Classical work in this area has spanned the theoretical foundations of algorithms for instruction scheduling with provable optimality, as well as heuristic approaches with experimentally validated performance improvements. Typically, the theoretical foundations are developed in the context of basic-blocks of code. In this paper, we provide the theoretical foundations for scheduling basic-blocks of instructions* with time-constraints*, which can play an important role in compile-time ILP optimizations in embedded applications. We present an algorithm for scheduling unit-execution-time instructions on machines with multiple pipelines, in the presence of precedence constraints, release-times, deadlines, and latencies l_{ij} between any pairs of instructions i and j. Our algorithm runs in time $O(n^3\alpha(n))$, where $\alpha(n)$ is the functional inverse of the Ackermann function. It can be used construct feasible schedules for two classes of instances: (1) one pipeline and the latencies between instructions are restricted to the values of 0 and 1, and (2) arbitrary number of pipelines and monotone-interval order precedences. Our result can be seen as a natural extension of previous work on instruction scheduling for pipelined machines in the presence of deadlines.*

Keywords: *Compiler-optimizations, embedded applications, instruction level parallelism, instruction scheduling.*

*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

Rooted in classical deterministic scheduling principles, *instruction scheduling* is a key optimization ubiquitous to modern compilers. Instruction scheduling algorithms attempt to minimize the overall *makespan* of a given set of instructions, as they execute on a RISC processor that embodies parallelism at a fine-grained level. Typically, such a processor can issue multiple instructions in a single cycle on multiple pipelined functional units. In particular, the core case of instruction-scheduling for *basic-blocks* is very well-understood and provably optimal solutions and heuristics [1, 10, 16, 22] have been validated in "product-quality" compilers.

With the ever-decreasing cost and improving performance of modern RISC microprocessors, they 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, these embedded applications often require that specific timing relationships to hold between the instructions. For example, some of the instructions might have real-time constraints specified as *deadlines* and *release-times*.

In this paper, we describe the first polynomial time algorithm that is guaranteed to find feasible schedules given basic-blocks of code with these real-time constraints, whenever such schedules exist, on various models of RISC machines. In doing so, we unify and generalize the earlier work due to Palem and Simons [16] on instruction scheduling with deadlines, and that of Garey and Johnson on scheduling tasks with release-times and deadlines on two identical processors [8]. Running in $O(n^3\alpha(n))$ time, our algorithm is guaranteed to find feasible schedules for arbitrary basic-blocks of code, for such RISC machines as the IBM-801 [18], the Berkeley RISC [13] and Stanford MIPS [11] 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* [16, 17]; in this case, the RISC processor can have a variable number m of RISC pipelines, each with k stages in it.

A crucial technical idea on which our algorithm rests is that of a *stable* and *consistent* modified deadline, that we associate with each instruction. The major algorithmic contribution in this paper is a fast algorithm 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 (Section 4.3) in just $O(n)$ iterations.

Feasible schedules are then computed by list-scheduling the instructions in non-decreasing order of their modified deadlines. 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 [10, 22]. The proof that for problem classes of interest, list-scheduling using our modified deadlines always yields feasible schedules whenever such schedules exist, is sketched in Section 5.

In Section 3, we relate our work with previous research in the rich area of deterministic scheduling. As sketched there, an interesting aspect of our work is that most of the previously known problem classes solvable in polynomial time can be re-derived using our modified deadline algorithm as special cases.

2. The Problem Definition

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. In this paper we are mainly concerned with *u*nit *e*xecution *t*ime (uet) systems, where $p_i = 1$. 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 (written as $i \prec j$) might have to be delayed an extra l_{ij} cycles after i completes, before it can start executing. Here, the function $l_{ij} \geq 0$ is referred to as the *inter-instructional latency* between i and j, or latency for short; it is represented as a label associated with edge (i, j) of the graph. Note that the latency is assumed to be a function of both the producer and consumer instructions.

Let $\mathbf{I}$ be the set of n instructions to be scheduled. The feasible schedule time of an instruction i can be further constrained by individual release-times and deadlines, which we denote as r_i and d_i. The *predecessors* of i 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, and denote this as $i \parallel j$. We denote the set of instructions independent of i as $indep_i \stackrel{\text{def}}{=} \{j \in \mathbf{I} \mid i \parallel j\}$.

With these definitions, the **Time-Critical Scheduling** problem that we will consider can be phrased formally as follows:

INSTANCE: Given a set of instructions $\mathbf{I}$, a partial order $\prec$ on $\mathbf{I}$ that encodes the dependences, 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. **precedence and latency**: for all $i \prec j$, $\sigma(i) + l_{ij} + 1 \leq \sigma(j)$
2. **resource**: for all time t, $|\{i \in \mathbf{I} \mid \sigma(i) \leq t < \sigma(i) + 1\}| \leq m$
3. **release-times**: for all $i \in \mathbf{I}$, $r_i \leq \sigma(i)$, and
4. **deadlines**: for all $i \in \mathbf{I}$, $\sigma(i) + 1 \leq d_i$.

In the notation of [9] and [2], this is the problem

$$P|prec(l_{ij}), r_i, p_i = 1|L_{\max}$$

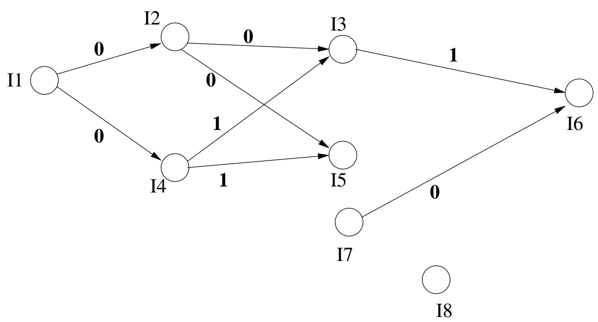

Figure 1. An example DAG

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 on time-step 2 due to its latency with instruction $I4$; it must wait an extra-cycle after $I4$ completes during step 2.

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

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

Now, 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 $j \in \{3, 4, 6, 7\}$. We say that instruction i *meets (misses)* its deadline if $\sigma(i) < d_i$ ($\sigma(i) \geq d_i$). The schedule in Figure 2 is no longer feasible with these new constraints since $I2$ misses its deadline. From the constraints it can be inferred that instruction $I2$ must be scheduled in step 2 in *any* feasible schedule. A feasible schedule S_2 for this problem is displayed in Figure 3. It also happens to be optimal.

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

Figure 3. Feasible time-constrained schedule S_2.

One particular important type of precedence order is an *interval-order*. 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 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 $pred_j \subseteq pred_{j'}$.

In keeping with standard assumptions in this field, we assume that the precedence order is given in *transitively closed* form. Furthermore, we assume that the maximum latency $l^+(i, j)$ between all pairs of nodes i and j is given as the input; the latencies and the processing times serve as the edge and node weights. If these values are not preprocessed, they can be computed within our stated time-bounds via matrix multiplication. For convenience, we denote $l^+(i, j) = -\infty$ if $i \not\prec j$.

3. Summary of Results

We present an algorithm that is guaranteed to find a feasible time-constrained schedule for unit-processing-time instances whenever such a schedule exists in $O(n^3\alpha(n))$ time, given

1. arbitrary data-dependence graphs ($\prec$ can take any form), as well as arbitrary release-times and deadlines, provided the latencies l_{ij} are either zero or one, and there is a single functional unit ($m = 1$). These restrictions on the latencies model early RISC processors such as the IBM-801, Berkeley RISC and Stanford MIPS processors [10, 22].

2. for DAGs that are monotone interval-orders, and on machines with an arbitrary number m of pipelines.

The running times of the algorithm can be improved to $O(n^3)$ in both cases using the techniques identified by Gabow and Tarjan [7], and by Frederickson [6], using possibly more than $O(n^2)$ space[1].

Interestingly, our algorithm also unifies several previously known results involving latencies, deadlines and release-times. 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 [3]; 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 [15] 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 [16], as well as those due to Bernstein and Gertner [1] for deadline-free instances. Again, release-times are not considered.

3. Moving to the third group, with release-times, our algorithm can be specialized to the case where all the latencies are zero, and $m = 2$ identical processors; the algorithm for this case is due to an early result by Garey and Johnson [8], which runs in $O(n^3)$ time.

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

[1] 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.

1. We can generalize the recent result due to Finta and Liu [5] and show that modified deadlines computed with our algorithm can be used with list-scheduling to minimize the makespan of a schedule on one processor with arbitrary precedence constraints, individual processing times $p_i \geq 1$ and zero or unit latencies; the work in [5] 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 [4], again via the unification in [15].

4. Computing the Modified Deadlines

The modified deadlines computation algorithm we describe below repeatedly invokes a *backward scheduling* subroutine backschedule, which is a generalization of the backward scheduling process described in Palem and Simons [16].

4.1. A Backward Schedule

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."

A schedule σ' such that the completion time of i is maximized is called a *backward schedule* of i. Note that instruction i precedence and latency constraints between instructions from the set S are ignored within the subroutine backschedule. 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 original problem. During the modified deadlines computation, we repeat this revision until no better estimate can be found.

4.2. Computing the Backward Schedule Efficiently

We solve the backward scheduling problem by reducing it into a sequence of simple decision questions. The reduction goes as follows. Iterate from the time step $t = d'_i - 1$ down to $t = \max\{d'_i - 2n, r'_i\}$. Lemma 3 shows that a feasible backward schedule can be found within this range.

For each time step t, we fix instruction i at time t in the schedule σ'. The latency constraints between i and $j \in S$ can be encoded as a new set of release-times for j. The problem then simplifies to a *decision* problem of scheduling unit-time tasks with release-times and deadlines (without precedence and latency constraints). This well-known problem can be easily solved by list-scheduling with Jackson's rule [12]: i.e. schedule a ready instruction with the earliest deadline whenever possible.

Since this 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 [20] data structure in $O(n\alpha(n))$ time and $O(n)$ space[2].

Our fast backward scheduling algorithm is described below.

Algorithm 1 (Backward Schedule backschedule**)**

1. Start from $t = d'_i - 1$, and search backwards one time step at a time until $t = \max\{d'_i - 2n, r'_i\} - 1$ is reached.

2. For each time step t, reduce the problem to a latency-free decision problem with only release-times and deadlines as follows. For each $j \in S \cup \{i\}$, compute a new set of release-times $\hat{r}$ and deadlines $\hat{d}$ using the following reductions:

$$\hat{r}_j := \begin{cases} \max\{r'_j, t + 1 + l_{ij}^{+}\}, & \text{if } j \in S \\ t, & \text{if } j = i \end{cases}$$
$$\hat{d}_j := \begin{cases} d'_j, & \text{if } j \in S \\ t + 1, & \text{if } j = i \end{cases}$$

3. If $t = d'_i - 1$, construct the lists L_r and L_d by sorting on the keys $\hat{r}$ and $\hat{d}$ respectively. Otherwise, reconstruct the lists L_r and L_d from the old lists computed in the last iteration. This reconstruction can be realized in time $O(n)$ as follows:

   (a) Scan the old list L_r and partition it into two lists L_1 and L_2. List L_1 contains all the instructions

[2] We omit the details of this improvement due to space limitation.

j whose release-time $\hat{r}_j$ is unchanged from the last iteration, while L_2 contains all the instructions whose release-time $\hat{r}_j$ is reduced by one from the last iteration. The new sorted list L_r can be constructed by merging the two sorted lists L_1 and L_2.

(b) 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 the list-scheduling algorithm on the inputs $\{i\} \cup S$, $\hat{r}$, $\hat{d}$, L_r, and L_d.

5. If list-scheduling returns "feasible," return $t + 1$ as the maximal completion time for instruction i. Otherwise, decrement t and repeat from Step (2) until $t < \max\{d'_i - 2n, r_i\}$.

6. If no feasible t exists, return "infeasible."

4.3. Fast Modified Deadline Computation

In general, any modified deadline computation scheme that invokes the backschedule subroutine should have two essential properties: *stability*[3] and *consistency*, as defined below.

Definition 1 (Stability) *Let d' be the modified deadlines computed from some iterative scheme $\mathcal{A}$. Then we say that d' is* stabilized *iff $d'_i = \mathsf{backschedule}(i, \mathbf{I} - \{i\}, r', d')$. 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 (Consistency) *Let r' and d' be the modified release-times and deadlines computed. Then we say r' and d' are* consistent *iff in all feasible schedule σ, $r_i \leq r'_i \leq \sigma(i) < d'_i \leq d_i$.*

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 backschedule subroutine; while consistency guarantees that whenever a feasible schedule σ exists for an instance, σ always satisfies the modified release-times and deadlines, and vice versa.

A naive method of computing a stable and consistent set of modified deadlines is to apply the backward scheduling primitive on each instruction i, and update d'_i with

$$\mathsf{backschedule}(i, \mathbf{I} - \{i\}, r', d')$$

[3] This is also called internal consistency by some.

until d' converges. It can be shown that any iteration order that visits all instructions must converge in at most n^2 iterations.

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 presented as algorithm modified_deadlines below.

Algorithm 2 (modified_deadlines)
INPUT: An instance of the unit processing time real-time scheduling 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[4] such that they are consistent with the partial order $\prec$ and the latencies: for all $j \in \mathbf{I}$,

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

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

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

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 backschedule 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 naive iteration scheme: this algorithm invokes the subroutine backschedule with only the previously processed elements as parameters.

4.4. An Example

Let us now test the algorithm modified_deadlines 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 process the instructions in the non-increasing order of r'_i, which is $I6, I7, I8, I5, I3, I4, I2$ and $I1$. Figure 4 displays the

[4] Update deadlines in reverse topological order, starting with instructions without successors.

backward schedules and the new modified deadlines computed in each invocation of the backschedule subroutine. 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.

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

Figure 4. Backward schedules and modified deadlines computed with Algorithm 2

By list-scheduling the instructions in non-decreasing order of their modified deadlines, we can obtain the optimal schedule in Figure 3.

4.5. Algorithmic Complexity

It is sufficient to invoke the list-scheduling subroutine at most $O(n)$ times in each invocation of backschedule. The main intuition is that starting from the modified deadlines initialized in Step (1) and subsequently in Step (4) of Algorithm 2, the list-scheduling subroutine must return a maximal completion time for i that deviate at most $2n$ time steps from the initial estimate, assuming that a feasible schedule exists.

Lemma 3 *In Algorithm* backschedule*, starting from modified deadlines that are consistent with the latencies, we need to iterate over at most $2n$ time steps to find a feasible backward schedule, or else no feasible schedule exists.*

Theorem 4 *Algorithm 1 runs in time $O(n^2\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function.*

Proof: By lemma 3 it is sufficient to invoke the list-scheduling algorithm$O(n)$ times in Algorithm 1. The list-scheduling algorithm uses 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 lists L_r and L_d are constructed by sorting when $t = d'_i - 1$, which takes a total $O(n \log n)$ time. When $t < d'_i - 1$, the lists L_r and L_d can be reconstructed from the old lists computed in the last iteration. This can be done in $O(n)$ time as described in Step (3).

Each run of the list-scheduling subroutine takes $O(n\alpha(n))$ time. Therefore, the total running time of Algorithm 1 is $O(n)O(n\alpha(n)) + O(n \log n) = O(n^2\alpha(n))$

Theorem 5 *Algorithm 2 (*modified_deadlines*) runs in time $O(n^3\alpha(n))$ and space $O(n^2)$.*

The algorithm assumes that the input DAG is given in transitively closed form. Since the transitive closure of l, l^+, can be computed in $O(n^3)$ time and $O(n^2)$ space, inclusion of the transitive closure phase in our analysis does not affect the time and space complexity of our algorithm.

The time complexity of our algorithm can be further improved to $O((m + n)n^2)$, where m is the number of pipelines, using Frederickson's [6] algorithm for scheduling unit-time tasks with integer release-times and deadlines, which can be made to run in $O(n)$ time using the off-line union-find algorithm of Gabow and Tarjan [7]. 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 Algorithm 2 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.

Lastly, schedules with minimal tardiness can be constructed using binary search by using our algorithm as a "black box." This adds a logarithmic overhead to the time complexity.

5. Correctness

We will now show that given a feasible instance, Algorithm modified_deadlines converges to a set of stable and consistent modified release-times and deadlines. Then in Section 6 we will show that the modified deadlines computed via our method can be used in conjunction with list-scheduling to find a feasible schedule when one exists.

First, a technical lemma about the structure of backward schedules is needed.

Lemma 6 *Let i be any instruction. Let r' and d' denote any modified release-time and deadline assignments. Suppose $t =$ backschedule$(i, succ_i \cup indep_i, r', d')$. Define the* supporting instructions *for i as the set*

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

Then

$$\begin{aligned} t &= \mathsf{backschedule}(i, succ_i \cup indep_i, r', d') \\ &= \mathsf{backschedule}(i, S(i, r', t), r', d') \end{aligned}$$

One consequence of this lemma is that we can compute the modified deadline for an instruction i by considering only its (potential) supporting instructions at each step. A graphical illustration of this lemma is presented in Figure 5. Here, a backward schedule for i is represented as a timing diagram. We use lightly shaded squares to represent idle slots or time slots scheduled with instructions with release-times less than t, and dark shaded squares to represent time slots scheduled with either successors of i or instructions with release-times greater or equal to t. The lemma claims that all the lightly shade squares can be replaced by idle slots and we can obtain the same modified deadline for i, i.e. the maximal completion time for i in both backward scheduling problems are identical.

Using the above lemma, we can show that the modified deadlines computed by Algorithm modified_deadlines are stable.

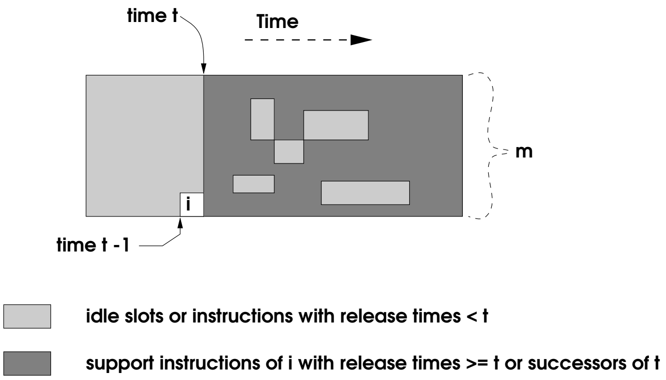

Figure 5. The backward schedule for i where backschedule$(i, succ_i \cup indep_i, r', d') = t$**.**

Lemma 7 *Algorithm* modified_deadlines *computes stable modified release-times and deadlines, i.e. for all instruction* i*,* $d'_i =$ backschedule$(i, succ_i \cup indep_i, r', d')$*, or else the instance is infeasible.*

Proof: We assume that the instance is feasible. We will show that only stable instructions are removed from the set U during Step (3) of the algorithm.

Let i be the first instruction chosen in Step (3). The initialization of r' in Step (1) guarantees that $j \prec i$ implies $r'_j < r'_i$. From this, we know that i cannot have any successors. Since $r'_j \leq r'_i$ for all $j \in \mathbf{I}$, by lemma 6 j cannot be a supporting instruction of i. Thus $d'_i = d_i$, and i must be stable after this step and can be removed from U.

Now assume that all $K \geq 1$ instructions that have been removed from U are stable. Let i be the next instruction to be chosen. By the choice of i all the remaining instructions $j \in U$ must satisfy $r'_j \leq r'_i$. Thus all the instructions in U are either predecessors of i or instructions independent of i. Consider any instruction $j \in U$. Since $r'_j \leq r'_i$, and the instance is feasible, we have $r'_j \leq r'_i < d'_i$. Thus j cannot be a supporting instruction for i in *any* subsequent iteration and thus by lemma 6 cannot affect the modified deadline of i Thus i is stable after Step (3).

By induction, the lemma holds.

The modified deadlines computed by Algorithm modified_deadlines are consistent, as stated in the following lemma.

Lemma 8 *Algorithm* modified_deadlines *computes consistent modified release-times and deadlines. Algorithm 2 returns "infeasible" only if the instance is infeasible.*

Proof: It is sufficient to see that each invocation of the backschedule algorithm is solving a relaxed instance of the original scheduling problem.

6. Optimality

Theorem 9 *Using the modified deadlines computed in Algorithm 2, list-scheduling produces a feasible schedule iff one exists, when restricted to the following classes of instances:*

(I) one pipeline and the l_{ij} are restricted to 0 or 1 $(1|prec(l_{ij} \in \{0,1\}), p_i = 1, r_i|L_{\max})$,

(II) arbitrary number of pipelines and the precedence constraints and latencies form a monotone interval-order graph $(P|$monotone–interval$(l_{ij}), p_i = 1, r_i|L_{\max})$,

(III) two processors and latencies are restricted to 0 $(P2|prec, p_i = 1, r_i|L_{\max})$, and

(IV) in-trees precedences, l_{ij} is equal to some constant l, and no individual release times $(P2|intrees(l_{ij} = l), p_i = 1|L_{\max})$.

Algorithms and optimality proofs for classes *(III)* and *(IV)* have previously appeared in [8] and [3]. We repeat the proof for *(III)* here for comparison.

Proof Sketch: Suppose we are given a feasible instance but list-scheduling produces an infeasible schedule σ. Let instruction i be an instruction with $\sigma(i) \geq d'_i$ such that $\sigma(i)$ is minimized. 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 scheduled with m saturated instructions. Let t_u be the latest unsaturated time step less than $\sigma(i)$[5]. 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 with release times that is less than or equal to t_u. Since t_u is unsaturated and

[5] We let t_u to be -1 if there are no unsaturated time steps before $\sigma(i)$.

since list-scheduling is greedy[6], we can infer that for all instructions $j \in \Sigma'$, there must exist a predecessor k such that $\sigma(k) + l^+_{kj} + 1 = \sigma(j)$. Now consider a few cases.

Case 1 If $|\Sigma'| = 0$ then by the pigeon hole principle no feasible schedule exists since we have to pack $|\Sigma| = m \times (\sigma(i) - t_u - 1) + 1$ instructions into $\sigma(i) - t_u - 1$ time steps on m pipelines. This is a contradiction.

Case 2 We assume $|\Sigma'| \geq 1$. We decompose our analysis into three cases, each corresponding to a different class of instances. Each of these cases share the following surprisingly structural similarity: in each case we are able to locate a critical instruction which is a predecessor of all the instructions in Σ'. This instruction is in some sense the bottleneck of the schedule.

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.

Case 2.3 *(III)* Since latencies are restricted to 0 and the number of processors to two, there must be one saturated instruction k scheduled at time t_u and it must be a predecessor of all instructions in Σ'. This implies that k must have missed its modified deadline, from the way the backward scheduling is performed, a contradiction.

7. Instructions with Individual Processing Times

Finally, modified deadlines computed with our algorithm can also be used to construct schedules with optimal makespan for a single processor, when instructions can have individual processing times of one or more, and in the presence of precedence and latencies of 0 and 1 ($1 \mid prec(l_{ij} \in \{0, 1\}), p_j \geq 1 \mid C_{\max}$). 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 [21].

A similar problem with unit-latencies, $1 \mid prec(l_{ij} = 1), p_j \in \mathbf{N}_+ \mid C_{\max}$, has been addressed in [5], using the Coffman/Graham [4] lexicographical rank function for list-scheduling priorities. (See also the recent work of Brucker and Knust [2] for other related results.) 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 any 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.

Theorem 10 *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

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

Note that the result depends only on the existence of an optimal scheduling algorithm for the unit processing time problem. Thus a faster algorithm exists. The Bernstein/Gertner [1] rank function can be computed in $O(n^2)$ time and is optimal for this problem. A final remark is that our result does not depend on the fact that the processing times p_i's are integral, and thus any arbitrary real processing times ≥ 1 will suffice.

8. Remarks

Decision questions arising out of parametric variations of the polynomially solvable instruction-scheduling variations for RISC machines and pipelines that we identified in Section 3 are known to be NP-complete [3, 10, 15]. Some interesting and challenging open questions that remain unresolved in this domain and that arise out of the new work are:

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

1. Designing a fast heuristic to find feasible schedules for tasks with unequal processing times and latencies that are either zero or unity, with release-times and deadlines. This case has the potential for being quite useful in practice.

2. Still staying with the makespan minimization version, typical approximation results use extensions of the classical algorithm of Graham [16] and tend to approach a constant factor ratio of 2 from below, as the problem parameters such as the number of functional units m increase. It will valuable to adapt powerful techniques rooted in polytope rounding [14] and related methods (for a recent update please also see [19]) to design approximation algorithms that can give better ratios in the context of RISC pipelines, for instructions with arbitrary precedence constraints.

3. Designing fast approximation algorithms, ideally with (small) constant fraction ratios, within the context of tasks with deadlines. Here the field is wide open and even the notion of a good objective function to approximate is not known. For example, the natural objective function namely *tardiness* which is defined to be the maximum amount by which any task exceeds its deadline, can assume a value of zero; in this case, any approximation ratio would tend to ∞. The questions raised in item 2 above all become relevant once again, should a meaningfully approximable objective function be introduced[7].

4. All of the above questions, within the context of time-constraints that are richer than fixed release-times and deadlines. An example is to consider bounds on the differences between the times that instructions i and j are executed. Examples are constraints of the form $\sigma(i) - \sigma(j) \leq c$ for a constant c.

Finally, we are planning on evaluating our algorithms in a working compiler.

References

[1] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, 1989.

[2] P. Brucker and S. Knust. Complexity results for single-machine problems with positive finish-start time-lags. *Osnabruecker Schriften zur Mathematik, Reihe P, Nr. 202*, 1998.

[3] J. Bruno, J. Jones, and K. So. Deterministic scheduling with pipelined processors. *IEEE Transactions of Computers*, C-29:308–316, Apr. 1980.

[7] We do not consider the measure $(tardiness + 1)$ as satisfactory although it remains finite everywhere.

[4] E. Coffman and R. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[5] L. Finta and Z. Liu. Single machine scheduling subject to precedence delays. *Discrete Applied Mathematics*, 70:247–266, 1996.

[6] G. N. Frederickson. Scheduling unit-time tasks with integer release times and deadlines. Technical Report CS-81-27, Dept. of Computer Science, Penn. State University, 1982.

[7] H. N. Gabow and R. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Proc. of ACM Symposium on Theory of Computation*, pages 246–251, 1983.

[8] M. Garey and D. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal of Computing*, 6(3):416–426, Sept. 1977.

[9] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

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

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

[12] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical Report 43, Management Science Research Project, University of California, 1955.

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

[14] J.-K. Lenstra, D. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Proceedings of the 28th IEEE Conference on Foundation of Computer Science*, pages 217–224, 1987.

[15] K. Palem. On the complexity of precedence constrained scheduling. Technical Report TR-86-11, University of Texas, Austin, TX, 1986.

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

[17] C. Papadimitriou and M. Yannakakis. Scheduling interval-ordered tasks. *SIAM Journal of Computing*, 8:405–409, 1979.

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

[19] A. Schulz. *Polytopes and Scheduling*. PhD thesis, Fachbereich Mathematik, Technische Universität Berlin, Berlin, Germany, 1996.

[20] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, April 1975.

[21] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[22] H. Warren. Instruction scheduling for the IBM RISC system/6k processors. *IBM Journal of Research and Development*, pages 85–92, 1990.