Highly Efficient Instruction Scheduling of Real-time Programs on RISC Processors*

Allen Leung
Courant Institute
of Mathematical Sciences
`leunga@cs.nyu.edu`

Krishna V. Palem
Courant Institute
of Mathematical Sciences
`palem@cs.nyu.edu`

Amir Pnueli
The Weizmann Institute
of Science
`amir@wisdom.weizmann.ac.il`

Abstract

Enabled by RISC technologies, low-cost commodity microprocessors are performing at ever increasing levels, significantly via *instruction level parallelism (ILP)*. This in turn increases the opportunities for their use in a variety of day-to-day applications ranging from the simple control of appliances such as microwave ovens, to sophisticated systems for cabin control in modern aircraft. Indeed, "embedded" applications such as these represent segments in the computer industry with great potential for growth. However, this growth is currently impeded by the lack of robust optimizing compiler technologies that support the assured, rapid and inexpensive prototyping of real-time software in the context of microprocessors with ILP. In this paper, we will present fast (polynomial-time) algorithms for compile-time *instruction scheduling*, of programs instrumented with *timing-constraints*, on processors with ILP. Our algorithms can be distinguished from earlier work in that they are guaranteed to find feasible schedules — those satisfying the timing-constraints — whenever such schedules exist, in cases of practical interest. Consequently, they can serve as powerful engines and can simultaneously support the "analysis" of the program prior to compilation, as well as during compilation once a feasible schedule is identified via analysis. We will also describe a novel notation, *Time_tract*, for specifying timing-constraints in programs, *independent* of the base language being used to develop the embedded application; Time_tract specifications are language independent and can be instrumented into imperative and object-oriented languages non-intrusively. As we will show, the instruction scheduling questions that arise out of Time_tract specifications are always "tractable". In contrast, a range of specification mechanisms proposed earlier yield substantially intractable instruction scheduling questions, thereby limiting their potential utility. We will sketch a formal and precise comparison of the tractability and related expressive power issues between Time_tract and some of the extant mechanisms for specifying properties of timed programs; this will be done using the canonical framework of timed-automata.

New York University
Computer Science Department
Technical Report Number 723

*Supported in part by an award from Hewlett-Packard Corporation, from IBM corporation, and an NYU Research Challenge Grant.

1 Introduction

The last decade has seen an explosion in the availability of low-cost microprocessors enabling a range of new applications. The embedded systems area wherein a microprocessor executes a control function such as in a hand-held video game or in a cardiac arrythmia monitor, represents an area with great potential for growth. A significant part of this growth is spurred by innovations centered around RISC technology [42, 48], *instruction level parallelism (ILP)* being a notable example [47], [44]. These "multiple-issue" RISC architectures rely crucially on high-quality optimizing compilers to unlock their performance potential. In particular, compile-time instruction scheduling is a singular example of such an optimization [39] ubiquitous to any modern optimizing compilers targeting an ILP processor.

A significant hurdle to the extensive use of ILP processors in the context of embedded applications is the lack of automatic tools that support the rapid-prototyping of these applications [48]. Notably, these tools should be able to enforce *timing constraints* between specific program "events" and hence the corresponding instructions. Currently, much of this development is done by hand, with the programmer typically designing the program in a language such as C (or possibly C++ to some extent), tuning and running the program to see if the informally specified timing relationships are satisfied. This process is repeated till a satisfactory execution is found. While this is quite arduous even for "single-issue" processors, it is virtually impossible to consider without some form of automatic support, when multiple instructions are issued in a single cycle. *This difficulty is a key limitation in the use of ILP platforms in the embedded systems domain, motivating the need for launching a substantial software development effort in this direction* [48].

Conventional tools and optimizations are not very applicable to this task since, for example, they restructure the program quite substantially thereby changing their (programmer) intended order of execution. This makes typical optimizations untrustworthy to developers of embedded programs. Changing this situation and making optimizing compiler technology accessible and applicable is the key to increasing the use of ILP processors within the context of embedded system development.

In this paper, we take a concrete step in this direction and provide algorithms for instruction scheduling programs, instrumented with *timing constraints*. An important aspect of our approach is that we provide a novel notation *Time_tract* for specifying timing relationships between program "points" [1]. Both our instruction scheduling algorithms as well as the introduced specification notation are rigorously presented. The correctness of the algorithms is established by rigorous proofs, and the Time_tract specification language is given a precise semantics. For loop-free programs, we establish a connection between the model considered here and the now canonical timed-automata [4]. Using this connection, we will characterize the expressive power of our notation, in comparison with previous proposals aimed at expressing timed properties of programs.

A significant property of Time_tract specifications are that they can be instrumented to encode timing relationships *independent* of the control-flow of the base program, which can be implemented in a conventional language such as C or C++ for example. In this regard, the results and algorithms presented in this extended abstract are independent of the base language used to develop the embedded application itself.

Furthermore, as we show here, the instruction scheduling problems derived from Time_tract specifications are amenable to solution by fast algorithms. Concretely, for the canonical case that model early RISC processors such as the IBM 801 [44], Berkeley RISC [30], and the Stanford MIPS [27] machine, we provide a fast algorithm that can *provably* find a feasible schedule whenever such a schedule exists, for arbitrary basic-blocks [1] of code and Time-tract specifications. We note that the provable results of this kind are known for instruction scheduling, only in the context of basic-blocks [40, 9]; however, as detailed in Section 7, previously known algorithms are applicable in the context where there are no time-constraints [9], or when very limited form of time constraints in the form fixed individual deadlines are associated with each instruction [40].

An interesting aspect of instruction scheduling algorithms for basic-blocks is that historically, in the absence of time-constraints, they have provided crucial building blocks that work very well in heuristics for scheduling global program regions beyond basic-blocks [15, 32, 39, 46]. We anticipate that this will be the case when we experiment with our algorithms presented here, in the context of scheduling beyond basic-blocks. As further evidence of the tractability of the global scheduling problems generated from Time_tract, we shown in this paper that they can be reduced in polynomial time to global instruction scheduling problems that are solved routinely by modern optimizing compilers, in the absence of time constraints. Based on this "evidence", we also propose extensions to Time_tract which share this property while offering a richer language for expressing time constraints.

1.1 Summary of Main Results

Let us now consider the main results presented in this paper.

1. We introduce a language-independent notation Time_tract for specifying timing constraints in programs (Section 2). These constraints permit placing absolute bounds on the actual times when specific program events occur; additionally, absolute and relative bounds on the "start-times" and "deadlines" for these events [18] can be specified.

2. We provide a fast instruction scheduling algorithm that runs in time $O(n^2 \log n + \Gamma n)$ given a basic block of n instructions and a Time_tract specification of size Γ.

   For machine models representing early RISC processors [30, 27, 44, 21] this algorithm will

   (a) provably find a "feasible" schedule relative to any Time_tract specification, whenever such a schedule exits,

   (b) and in $O(\log n)$ invocations, can find a schedule with minimum "tardiness" when a feasible schedule does not exist.

   Informally, the tardiness of a schedule is the maximum amount of time by which an instruction has exceeded its deadline. A feasible schedule has zero tardiness[1].

[1]Standard techniques will yield a schedule with minimum overall completion time among all feasible schedules, as a special case of minimizing tardiness.

Detailed definitions of instruction scheduling terminology can be found in Section 3.

3. In Section 4, we formulate the problem of global scheduling, whose complexity is analyzed in the next section.

4. We propose an extension to Time_tract referred to as *Extended Time_tract (ET_Tract)* that permits specification of relative time-constraints between the times when events occur; this has to be contrasted with Time_tract where such constraints can only be absolute. For the case of global scheduling, we also introduce the notion of *dynamic markers*. We show that the scheduling problems for acyclic program regions with ET_Tract specifications (and dynamic markers) are no harder than their counterparts[10, 39] which do not involve time constraints at all.

   We show (Section 5) that if we admit these more general type of constraints the instruction scheduling problems arising out of ET_tract specifications are NP-complete even for the simplest RISC-machine model discussed above.

5. Finally (Section 6), we introduce the notion of a *guarded command real-time program* (RTP, [13]), which is a model semantically equivalent to *timed automata* [4]. We show that the scheduling problem for basic blocks under ET_tract specifications is reducible to the reachability problem of RTP's. In general, this problem is PSPACE-complete, however for the class of RTP's obtained by a reduction from a basic-blocks scheduling problem (which we term *acyclic* RTP*'s*), the reachability problem is NP-complete. In the more general case of (possibly branching) acyclic regions scheduling under ET_tract specifications, we cannot reduce the problem to RTP-reachability and, instead, use a new problem: *the realizability problem* for RTP's. The complexity of this problem is the same as the reachability problem. That is, in the general case it is PSPACE-complete but, restricted to RTP's derived from an acyclic-regions scheduling problem, it becomes NP-complete.

In this extended abstract, we only consider loop-free programs. This will be extended in the full paper to programs with loops[2]

2 The Time_tract Specification Language

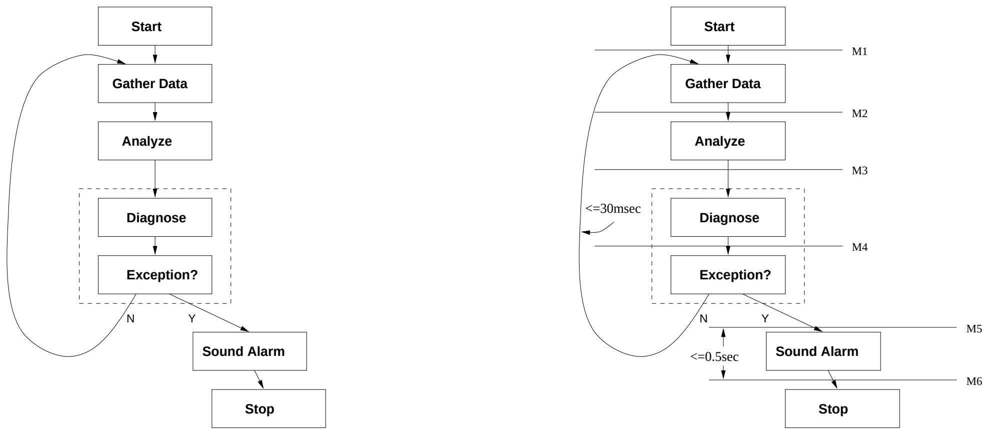
For convenience, we will first present the notions in the context of basic-blocks [1, 39] in this section. We will then build on this and discuss the context beyond basic-blocks in Section 4.

2.1 An Example Embedded Program

Consider an example program that controls a typical Cardiac Arrythmia monitor in a modern ICU [51]. In this case, we have a program $\mathcal{P}$ specified in an imperative or object-oriented language such as C or C++, as shown in Figure 1 (a) at the granularity of functional basic-blocks. The points of the program of interest are denoted by "markers" $M_1, M_2, \ldots, M_6$ (Figure 1 (b)), between which timing relationships are to be enforced. Markers are associated

[2] In particular, we will deal with the problem of periodicities and how are markers associated with program points within loops.

with "events" of interest; by convention, let us associate the event associated with marker M_i to be the execution of the instruction immediately following it[3].



(a) (b)

Figure 1: An Example Embedded Application Program

The two constraints of interest, stated informally here and comprehensively developed below require that:

1. any iteration of the loop to be completed in 30 msec and
2. whenever an abnormality (arrythmia) is detected, software sound an alarm in 500 msec.

Throughout this paper, we will assume that all our formal specifications refer to structured programs [1], and that all times are expressed in milliseconds.

2.2 Time_tract for Basic-blocks

Continuing with the example, let us consider the basic-block enclosed by markers M_5 and M_6. For illustration, we will consider this basic block in isolation in what follows, and will discuss its interaction with the rest of the program in Section 4[4]. Considering this block in isolation implies that, at time 0, we are ready to begin the execution of M_5.

Under this simplifying assumption, the constraint that the alarm be activated in 500 msec can be easily captured as constraints on the times τ_5 and τ_6 when the events associated with markers M_5 and M_6 respectively start. In this case, the constraints we need to specify are $\tau_5 \geq 0$ and $\tau_6 + \delta_6 \leq 500$ where δ_6 is the (possibly 0) duration of the event associated with marker M_6.

Equivalently, with each marker, we can specify a *start-time* s_i and a *deadline* d_i. Informally, the start-time indicates a lower-bound or an earliest time by which the event associated with M_i can occur, whereas the dual notion of a deadline indicates the latest time when this same event must complete; as we will see below, deadlines can be *absolute* or *relative*.

[3]We can accommodate alternate conventions which associate the execution of the instruction immediately preceding the marker to be its associated event.

[4]In fact, in this example application, all of the constraints can be evaluated strictly on basic-blocks.

Returning to the basic-block in the example of interest, the constraints between markers M_5 and M_6 bounding the delay in sounding the alarm can be encoded by absolute constraints as follows:

$$s_5 = 0; d_6 = 500$$

Since the constraints associated with M_5 and M_6 are both local to the enclosed basic-block, we have simplified the overall problem by associating a (virtual) start-time of zero with the start of this basic block via marker M_5. Now, requiring that this basic block execute in 500msec simply requires imposing an absolute bound on the deadline $d_6 \leq 500$ (from a start-time of 0 for M_5).

2.2.1 The Notation

As is standard, we assume that the program has a set a set of *program points* [1] $p_1, p_2, \ldots p_\lambda$ Thus, given a program point j, we have a well-defined notion of an instruction $(j-1)$ that precedes it and an instruction j that succeeds it[5]. A set of *markers* $M_1, M_2, \ldots M_\mu$ will be associated with some selected subset μ of the set of program points, which identify the points in the program associated with "events" of interest.

Timing relationship between markers are specified as a family of constraints $\mathcal{C}$ on the *scheduled-time* τ_i, *start-time* s_i and *deadline* d_i associated with marker M_i. For convenience, in what follows, we will directly associate constraints with instructions rather than markers without loss of generality, since there is one-one correspondence between them.

In general, the constraints in $\mathcal{C}$ can be absolute or relative as follows. Absolute constraints in Time_tract on instruction I_i are of the form $a \leq \tau_i \leq b$ for non-negative integers a and b. Or equivalently, we can also have constraints on the start-times and deadlines of the form $s_i \geq a$ and $d_i \leq b$ for non-negative integers a and b. Informally, this implies that the event associated with marker M_i must occur in the interval of time (a, b) where $a \leq b$. For convenience, we refer to constraints of this form as type-1 constraints.

Finally, we also permit relative constraints between the start-times of various markers, as well as the deadlines. The most natural case is one in which only one variable occurs on each side of the inequality, i.e., $s_j \geq s_i + a_{ji}$; similar constraints can hold between the deadlines as well. We note that in Time_tract, start-times are only related to other start-times, whereas deadlines are only related to other deadlines. We refer to these relative constraints as type-2 constraints.

Relationship Between Type-1 and Type-2 Constraints While type-2 constraints seem to permit a more general framework for specifying timing relationships, they are in fact equivalent to type-1 constraints. As shown in Section 5, we can always reduce a mixed system of type-1 and type-2 constraints of size Γ with n variables in $O(\Gamma n)$ time to an equivalent system of type-1 constraints alone.

Actually, this equivalence and all our instruction scheduling algorithms and related claims are valid even if the relative constraints are further extended as follows[6]. Consider constraints of the from $s_j \geq \alpha \cdot \mathbf{s} + \mathbf{a}$; $\alpha = \langle \alpha_{1j}, \ldots \alpha_{\mu j} \rangle$, $\mathbf{s} = \langle s_{1j} \ldots s_{\mu j} \rangle$ and $\mathbf{a} = \langle a_{1j}, \ldots a_{\mu j} \rangle$ are all

[5]As usual, we assume that all the n instructions/statements of the program are labeled from the set $1, \ldots n$.
[6]Except that the complexity while remaining to be in P, increases.

non-negative rational vectors of length μ, and $\alpha_j, a_j = 0$. Informally, this allows us to relate the start-time of M_i to a linear combination of that of the other markers. We can similarly have deadline constraints of the form $d_i \leq \beta \cdot \mathbf{d} + \mathbf{b}$.

3 Instruction Scheduling Algorithms

We are now ready to introduce our primary technical result, namely an algorithm with provably good properties for scheduling basic-blocks derived from programs annotated with Time_tract specifications. In the interests of ease of exposition, we will first develop the algorithm for the restricted case wherein instructions only have type-1 constraints only. Subsequently, in Section 3.4.2, we will use this as "black-box" with well-defined and proven properties, to solve inputs with type-2 Time_tract specifications as well.

3.1 An Example and Problem Definition

Let us consider the example basic-block shown in Figure 2 below, denoted as a directed acyclic graph (DAG). The nodes of this DAG are the set of instructions in the original program $\mathbf{I}$, with the edges denoting data-dependences [1, 39]. Let us consider scheduling it on a processor with a single pipelined functional unit, with two stages; examples include early RISC processors mentioned above [44, 30, 27]. A directed edge $\langle I_i, I_j \rangle$ implies that due to data-dependence, instruction I_i must complete executing on the processor before I_j can start (we denote this as $I_i \sqsubset I_j$). This is reflected in schedule S_1, wherein all data-dependence relations are obeyed.

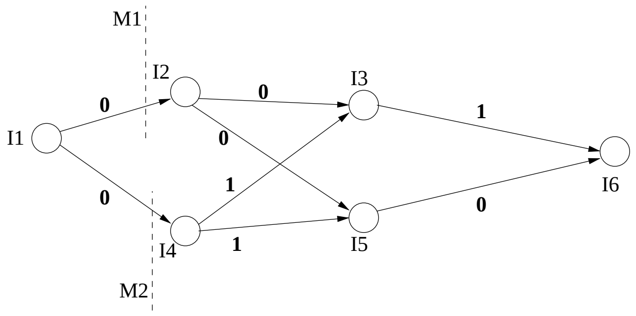


Figure 2: An Example Basic-block

In keeping with conventional modeling in this domain [9, 40], each instruction $I \in \mathbf{I}$ takes one cycle to execute. Additionally, load/store instructions take an extra cycle due to memory latencies[7]. This extra cycle is the *inter-instructional latency* associated on the edges of the DAG and specified as the function $w : \mathbf{I} \times \mathbf{I} \rightarrow \mathbf{N}$. As shown in Figure 2, $w(I_3, I_6)$, $w(I_4, I_3)$ and $w(I_4, I_5)$ are all equal to unity, whereas all other inter-instructional latencies are zero. Due to this latency, instruction I_6 cannot start till one cycle after I_3 completes (in one cycle). This is indicated by the idle cycle, denoted by ϕ, in step 7 of the schedule presented in Figure 3, when no other eligible instruction is available, and hence the pipelined functional unit is idle. Similar delays apply to instructions I_3 and I_5 caused by the latencies due to instruction I_4, during step 4.

[7]Here, memory latency refers to a cache hit; misses can result in substantially higher latencies.

Furthermore, in our example, since there is only one functional unit in the target, no more than one instruction can be scheduled on any cycle of the target; this is referred to as a *resource constraint* in scheduling parlance. In general, we can have num_l units of type l—types could be floating point, fixed point and branch prediction for example. Each node/instruction I is to be executed on a functional unit of a particular type specified via the function $type(I)$. Furthermore, a unit of type l can be pipelined to have $k_l + 1$ stages for a non-negative integer k_l. Now, give an instruction I such that $type(I) = l$, any edge starting at I (of the form $\langle I, I' \rangle$) can be labeled by an inter-instructional latency no greater than k_l.

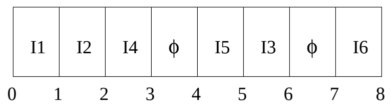


Figure 3: Example Schedule

Returning to the example with markers M_1 and M_2, we have absolute Time_tract constraints $s_2 = 1$ and $d_2 = 2$, as well as the relative constraint $d_1 \leq d_2$. The absolute constraints force instruction I_4 to be scheduled at time 2 in S_1. Given the resource constraint imposed by the single pipeline, we cannot schedule instruction I_2 at time 2. Therefore, in order to satisfy the relative constraint $d_1 \leq d_2$, the only choice is to schedule I_2 at time 1.

Additionally, for each marker say M_i in our example, we permit a start-time constraint s_i and a deadline constraint d_i. In case an instruction I is missing a start-time constraint, $s_I = 0$; similarly, if the deadline is missing, $d_i = s + (k+1).n$ where s is the largest of all the absolute start-times and k is the largest inter-instructional latency in the input[8]. We can also have a range of relative time constraints as stated before, between pairs of markers and $\mathcal{C}$ denotes the entire family of start-time and deadline constraints.

Given an instance of the scheduling problem specified as above, a *feasible schedule* is an assignment of time σ for initiating each instruction $I \in \mathbf{I}$ such that,

1. whenever $\langle I_i, I_j \rangle$ is defined and labeled by an inter-instructional latency $w(I_i, I_j)$, $\sigma(I_j) \geq \sigma(I_i) + w(I_i, I_j) + 1$;
2. the number of instructions with $\sigma(I) = j$ and $type(I) = l$ for all j and l is no more than num_l; and
3. there exists an assignment to s_i and d_i such that for all i, $s_i \leq \sigma(I_i) \leq d_i$ is satisfied. In section 3.4.2 we'll discuss efficient methods for separating $\mathcal{C}$ from the rest of the inequalities.

An initial objective of instructional scheduling is to find a feasible schedule such that $max\{\sigma(I)\}$ over all instructions $I \in \mathbf{I}$ is minimized; this gives us a feasible schedule with minimum overall completion time[9] Another objective of interest here and one which we can optimize

[8]Essentially, choose the deadline to be a reasonably large value by which the unconstrained instruction is guaranteed to complete.

[9]Note that dropping the time constraints will yield a schedule with minimum overall completion time—the goal of classical instruction scheduling [39].

using our algorithm is the *tardiness* of a schedule S. Let $Tar(I_i, S) = max\{(\sigma(I_i) + 1 - d_i, 0\}$ be the *tardiness* of instruction I_i *in* S. Then, the tardiness of S is the maximum value of tardiness over all instructions I_i in S.

Returning to the example DAG introduced above in Figure 2, we note that by interchanging the order of execution of instructions I_3 and I_5 in schedule S_1 (Figure 3), we get a feasible schedule S_2 with a smaller completion time as shown in Figure 4; it is easy to verify that S_2 has optimum completion time for the given example.

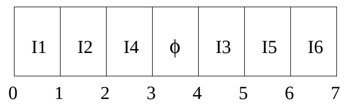


Figure 4: A Feasible Schedule with Minimum Completion Time

3.2 A Framework for Instruction Scheduling

In the past, a canonical framework has been developed for scheduling basic-blocks of instructions [26, 9, 39], and beyond [15]. This framework has the substantial advantage of being validated in practice in product quality compilers for RISC processors. Our algorithms also uses this framework for scheduling using Time_tract specifications.

The canonical framework has the following steps.

1. For each $I \in \mathbf{I}$, compute its *modified deadline* and rank the instructions in non-increasing order of their modified deadlines, with ties broken arbitrarily. Call the resulting (priority) list of instructions $\mathcal{L}$. Intuitively, the ranks will help distinguish the instructions so that those that are "likely to" help minimize the completion time and/or make the schedule feasible schedule are given priority; this step is the heart of our construction and Section 3.3 and 3.4 are dedicated to its description and properties.

2. Greedily *list-schedule* and construct a schedule $\sigma : \mathbf{I} \rightarrow \mathbf{N}$ from $\mathcal{L}$, as follows: An instruction I_i is termed *ready* at time t iff

   (a) $\sigma(I_j) + w(I_j, I_i) + 1 \leq t$ for all predecessors I_j of I_i, and

   (b) $s_i \leq t$

   For each successive time step t, list scheduling scans the priority list $\mathcal{L}$ looking for ready instructions. Each ready instruction I_i is assigned to a functional unit of type $type(I_i)$, as long as one remains available at the time step. Instructions are removed from the priority list as they are assigned, and this process repeats until all the instructions have been scheduled.

The challenge and novelty of our work is in constructing an appropriate rank function, which is also the most computationally intense part of this framework; we will consider this next.

3.3 Computing Ranks

Our approach to computing ranks for the instructions is via computing *modified deadlines* as detailed below; the modified deadlines will serve as ranks. Our proposed algorithm will compute modified deadlines for an arbitrary number of pipelines and arbitrary latencies. However, for simplicity we'll consider only the case where there are m identical pipelines and all instructions are of this type.

The auxiliary routine backschedule is used to construct a backward schedule. The ideas behind this subroutine and its description were first presented in [40]. The routine

$$\mathsf{backschedule} : \mathbf{I} \times 2^{\mathbf{I}} \times (\mathbf{I} \to \mathbf{N}) \to \mathbf{N}$$

takes an instruction I_i, a set of instructions $I' \subset \mathbf{I}$ which may affect the deadline of I_i (which include the successors of I_i and instructions independent of I_i) and a set of preliminary deadlines $\hat{d}$ on $\mathbf{I}$, and using this information, computes a new modified deadline for I_i. The backward schedule is constructed in the following manner: first, assign each instruction $I_j \in I'$ a lexical-graphical rank $(\hat{w}(I_i, I_j), \hat{d}_j)$ where $\hat{w}$ is defined as

$$\begin{array}{rcll} \hat{w}(I_i, I_j) & = & w(I_i, I_j) & \text{if } I_i \sqsubset I_j \\ \hat{w}(I_i, I_j) & = & -\infty & \text{otherwise} \end{array}$$

Notice that this definition of $\hat{w}$ generalizes that of [40]. We then construct a list L from I' by sorting the instructions in non-decreasing order of this rank, and proceed to schedule the instructions in the order of L with ties broken arbitrarily. For each $I_j \in L$, we assign to it the largest time step t such that (1) t is consistent with I_j's deadline, i.e. $t < \hat{d}_j$, and (2) no more than $m - 1$ previous instructions have been assigned to the same time step. The effect of the rank is that instructions of larger latencies from I_i are considered first (ties are broken by deadlines) and instructions independent with I_i are considered last. Finally, we assign to I_i the largest time step t such that t is consistent with the deadline of I_i and respecting the resource constraint as in (1) and (2) above, and moreover, (3) t is consistent with the precedence constraint and latencies, i.e. for all $j \in I'$, $t + \hat{w}(I_i, I_j) + 1 \leq \sigma(j)$ in the backward schedule. The new modified deadline of I_i is then $t + 1$.

Handling individual deadlines For simplicity, let us first suppose that the given problem instance only has absolute deadlines. We can obtain a set of modified deadlines using instructions I_i's successors (which we denote as the set $succ_i$) and their already computed modified deadlines as input. A *derived deadline* $\hat{d}$ is any function that assigns non-negative integer values to the instruction. A derived deadline is *consistent* provided that in every feasible schedule $\sigma(I_i) < \hat{d}_i$ and vice-versa.

That is, we compute the modified deadlines d'_i as

$$d'_i = \mathsf{backschedule}(I_i, succ_i, \hat{d})$$

where $\hat{d}$ is the modified deadlines of I_i's successors. This computation can proceed in reverse topological order of the precedence. Intuitively, the backschedule procedure attempts to schedule all the successors of I_i on the processors so as to ensure that they meet their (modified) deadlines. In order to satisfy this condition, it starts with the sink nodes (those without any successors) and works inwards in topological order. For example, it would start with $I6$ followed by $I3, I5, I2$ and $I4$. Intuitively, if an instruction I_i has a successor I_j that must complete by time t and the current deadline of I_i is greater than t then d_i can be "pushed forward" without changing the problem.

Instruction	Deadlines	Successors
I_1	7	$\{I_2, I_3, I_4, I_5, I_6\}$
I_2	4	$\{I_3, I_5, I_6\}$
I_3	6	$\{I_6\}$
I_4	5	$\{I_3, I_5, I_6\}$
I_5	8	$\{I_6\}$
I_6	8	$\{\}$

Figure 5: Example Deadlines for Basic Block.

For example, suppose we take the DAG presented in Figure 2 and impose on the instructions the deadlines in Figure 5. We can see that a possible processing order for the instructions is $I_6, I_5, I_3, I_4, I_2, I_1$. Given the resource constraint of one pipeline, Figure 6 illustrates the backward schedules constructed from running the **backschedule** subroutine.

Instruction	Modified Deadlines	Backward schedules							
		0	1	2	3	4	5	6	7
I_6	8								I_6
I_5	6						I_5	ϕ	I_6
I_3	5						I_3	ϕ	I_6
I_4	4				I_4	I_3	I_5	ϕ	I_6
I_2	4				I_2	I_3	I_5	ϕ	I_6
I_1	1	I_1	ϕ	I_2	I_4	I_3	I_5	ϕ	I_6

Figure 6: Example Modified Deadlines and Backward Schedules.

Modified deadlines computed in this manner have an interesting property: all feasible schedules that meet the original deadlines must also meet the modified deadlines. Stated formally, we can easily prove the following useful property:

Lemma 1 *Let* $d'_i =$ backschedule$(I_i, succ_i, \hat{d})$. *If* $\hat{d}$ *is consistent then a schedule* σ *meets all its original deadlines* d_i *iff all instructions* I_i *also meets the deadlines* d'_i.

Handling individual start-times and deadlines In the presence of both individual start-times and deadlines, the modified deadline computation scheme presented above will have to be generalized to take *independent* instructions into account. We say that two instructions I_i and I_j are independent iff $I_i \not\prec I_j$ and $I_j \not\prec I_i$. We denote this condition as $I_i \mid\mid I_j$ and set of instructions independent of I_i by $indep_i$. Intuitively, the deadline d_i of I_i have to made smaller in a schedule if some instruction I_j independent of I_i can only start after d_i. This can happen if there is not enough idle time slot after d_i.

Instruction	Modified Deadlines	Backward schedules							
		0	1	2	3	4	5	6	7
I_4	3			I_4	I_2	I_3	I_5	ϕ	I_6
I_2	4				I_2	I_3	I_5	ϕ	I_6
I_1	1	I_1	ϕ	I_4	I_2	I_3	I_5	ϕ	I_6

Figure 7: Example Modified Deadlines and Backward Schedules with $s_2 \geq 4$.

To illustrate, let's take the running example in Figure 2 with the deadlines presented in Figure 5. In addition, we impose the start-time constraint of $s_2 \geq 4$ to the problem while all other instructions are available at time 0. Observe that I_4 must be scheduled by time step 2 in all feasible schedules or else there will not be any available time slots remaining to schedule I_2. This fact is reflected in the enhanced backward schedule of I_4 constructed from the successors and instructions independent of I_i (Figure 7). It is easy to verify that the backward schedule constructed yields $d'_4 \leq 2$; contrast this with $d'_4 \leq 3$ in Figure 6 when no start-times are involved.

In general, in the presence of start-times, any pair of independent instructions I_i and I_j may exert "pressure" on each other forcing each other to be scheduled earlier in a feasible schedule. This phenomenon can be stated formally as follows. Define

$$d'(i,t) = \mathsf{backschedule}(I_i, succ_i \cup \{I_k \in indep_i | s_k \geq t\}, \hat{d})$$

and $\hat{d}$ is consistent as before.

Lemma 2 *If for some time step t, $d'(i,t) \leq t$ then I_i must meet the deadline $d'(i,t)$ in all feasible schedules.*

Proof: Define the modified deadline on i, d'_i as follows:

$$d'_i = \min_{t \in \mathbf{N}} \mathsf{backschedule}(I_i, succ_i \cup \{I_k \in indep_i | s_k \geq t\}, \hat{d})$$

Consider the deadline $r_i = \mathsf{backschedule}(I_i, succ_i, \hat{d})$. If $d'_i = r_i$ the theorem is immediate from lemma 1. Now suppose $d'_i < r_i$ and consider the backward schedule computed for $d'_i = \mathsf{backschedule}(i, S, \hat{d})$ (for some set S) and the backward schedule computed for r_i. We

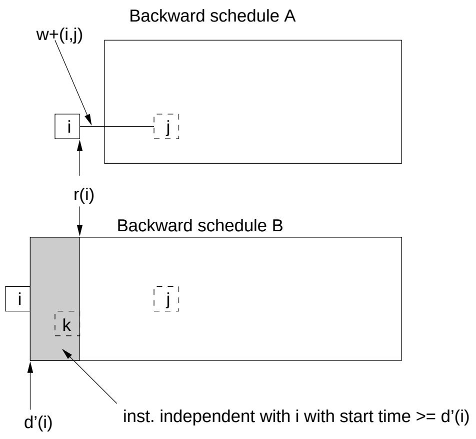


Figure 8: Proof of lemma 1.

know that the independent instructions $I_k \in S$ must be the last to be backward scheduled. Furthermore, if $d'_i < r_i$, the time slots between $[d'_i ... r_i)$ must be filled with these independent instructions. We know that for all $I_k \in S, i \mid\mid k$, the start-times $s_k \geq d'_i$, from the definition of set S. Figure 8 illustrates this phenomenon. In this figure, the backward schedule A corresponds to that of r_i and the backward schedule B corresponds to that of d'_i. The shaded region must contain only instructions I_k such that $I_i \mid\mid I_k$.

If I_i were to be pushed back in any schedule, then at least one of the instructions I_k in the time steps $[d'_i ... r_i)$ will have to be moved before d'_i, by a pigeon hole argument. But this will violate the start-time constraint of this instruction I_k. Thus the lemma holds. ■

This lemma extends the preliminary statement in lemma 1 which held only for deadlines, to include start-times as well.

Suggested Approach to an Algorithm The above lemma hints at an iteration scheme for computing the modified deadlines: iteratively compute $d'(i,t)$ for all i and t whenever the deadlines of $succ_i$ and/or $indep_i$ has been altered until all the modified deadlines become stable. We say that the modified deadline of an instruction has *stabilized* if it cannot be decreased further without violating the start-time constraint of some instruction. Instruction with stabilized deadlines are called *stable*. Clearly this process will terminate since all numbers involved are integral and bounded. However, this naive process may be very inefficient. Instead, we will organize the computation of modified deadlines in the following fashion, yielding a very efficient scheme while retaining this intuition.

Algorithm 1 (Modified Deadlines Computation)

1. Run the algorithm from [40] and compute the modified deadlines d' using the initial deadlines d. If in this step and in any other subsequent steps $s_i \geq d'_i$ for some i, we can stop immediate and conclude that no feasible schedule exists.

2. Sort the instructions in non-increasing modified deadline order and process the instructions subsequently in this order.

3. For each instruction I_i to be processed, compute the set of successors $succ_i$ and the list of independent instructions $indep_i$ sorted in the order of non-increasing start-times s_k.

4. (a) For each start-time t in the list of independents, compute the value of $d'(i,t)$ using the current modified deadlines as input. If for some t, $d'(i,t) \leq t$, set d'_i to be $\min(d'_i, d'(i,t))$.

   (b) Notice that step (a) can be computed efficiently by first constructing a backward schedule S_0 from only the successors $succ_i$, then incrementally add the new sets of independent instructions $\{I_k \in indep_i | s_k = t\}$ to S_0.

5. Repeat from steps (2) to (4) to verify that the deadlines have indeed been stabilized. If any of the deadlines change during this iteration also output "infeasible schedule."

6. Return the set of modified deadlines d'.

3.4 Main Properties of Our Algorithm

Let us start with the following technical lemma before establishing the most interesting properties of the algorithm.

Lemma 3 *Let* $\mathbf{s} \subset \mathbf{I}$ *be a set of stable instructions and let* I_i *be some instruction with the largest current modified deadline in the set* $\mathbf{I} - \mathbf{s}$. *Then* d_i *will be stabilized after running step (4) of algorithm 1 on* I_i.

Proof: Since $I_i \sqsubset I_j$ implies $d_i < d_j$, we know that all successors of I_i are stable. Consider the backward schedule(see Figure 9) that computes the new deadline upper-bound $d_{new}(i)$ with $d_{new}(i) \leq d_i$: i.e. for some t,

$$d_{new}(i) = \mathsf{backschedule}(I_i, succ_i \cup \{I_k \in indep_i | s_k \geq t\})$$

The backward schedule can contain three different types of instructions: (a) successors of I_i, all of which are stable; (b) instructions I_j independent of I_i that are stable; and (c) instructions I_j independent of I_i that are unstable and with start-times $s_j \geq d_{new}(i)$ and deadlines $d_j \leq d_i$. Furthermore, the backward schedule from time step at from $d_{new}(i)$ to $d_i + 1$ must be filled completely with instructions of type (b) or type (c): type (a) instructions cannot appear in this region due to the initialization done in step (1) of the algorithm. Of these, only the deadlines of type (c) instructions can be decreased. It can be seen with a pigeon hole argument that decreasing these deadlines cannot change $d_{new}(i)$ if a feasible schedule exists. Thus $d_{new}(i)$ has been stabilized.

■

Now assume that a feasible schedule exists. Since algorithm 1 always choose an unprocessed instruction I_i with the largest current modified deadline, by applying the above lemma inductively we know that all instructions are stable after they have been processed in step (4). Step (6) of the algorithm makes sure that the deadlines have indeed been stabilized: i.e. if any changes occur during this iteration, we can assume that from the definition of stability

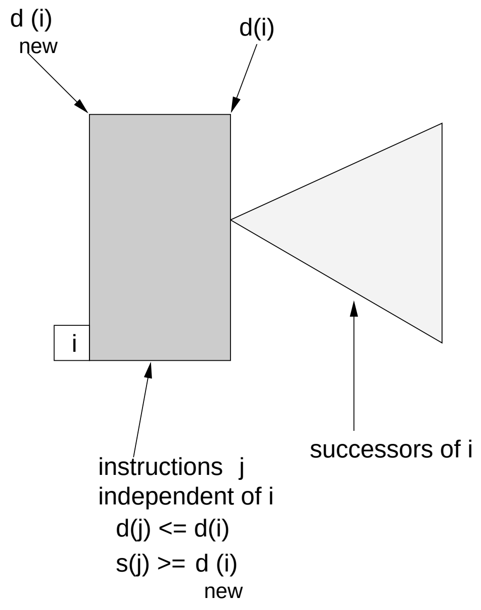


Figure 9: Backward schedule for i in the proof for lemma 3.

no feasible schedule exists. If this does not occur and stabilized deadlines are computed, we can then construct a priority list and perform list scheduling. We can further prove that this process is optimal if we are scheduling on one pipeline and (0,1)-latencies:

Theorem 4 *If there is only one pipeline and if the individual latencies are restricted to* $\{0, 1\}$*, the rank algorithm and list scheduling will find a feasible schedule iff one exists.*

Proof: Suppose that given a feasible instance of the problem the algorithm constructs an infeasible schedule σ. Since we know that the greedy list scheduling algorithm will not violate precedence, resource or start-time constraints, some instruction I_k must have failed its deadline constraint d_k. By lemma 2 we know that k will also have failed the modified deadline d'_k. Without loss of generality, consider the first instruction k in the schedule that fails its modified deadline. In particular, consider all the time steps from 0 to $\sigma(k)$.

We say that a time step t is *free* if it is idle or if the instruction I_i scheduled at the time step has a modified deadline $d'_i > d'_k$. Otherwise, we say the time step is *bound.* There are four cases to consider (see Figure 4[10]):

case 1 — The segment T from 0 to $\sigma(I_k)$ is bound (i.e. scheduled with instructions such that for all $I_i \in T$, $d'_i \leq d'_k$). By a pigeon hole argument we can show that at least one of $I_i \in T$ will miss its modified deadline in all schedules, contradicting the assumption that I_k is the first instruction to miss its deadline.

case 2 — The segment T from 1 to $\sigma(I_k)$ is bound but time step 0 is free. This can only happen in the greedy schedule if for all $I_i \in T$, $s_i \geq 1$. By a similar pigeon hole argument

[10] I_k is notated as simply k in this figure.

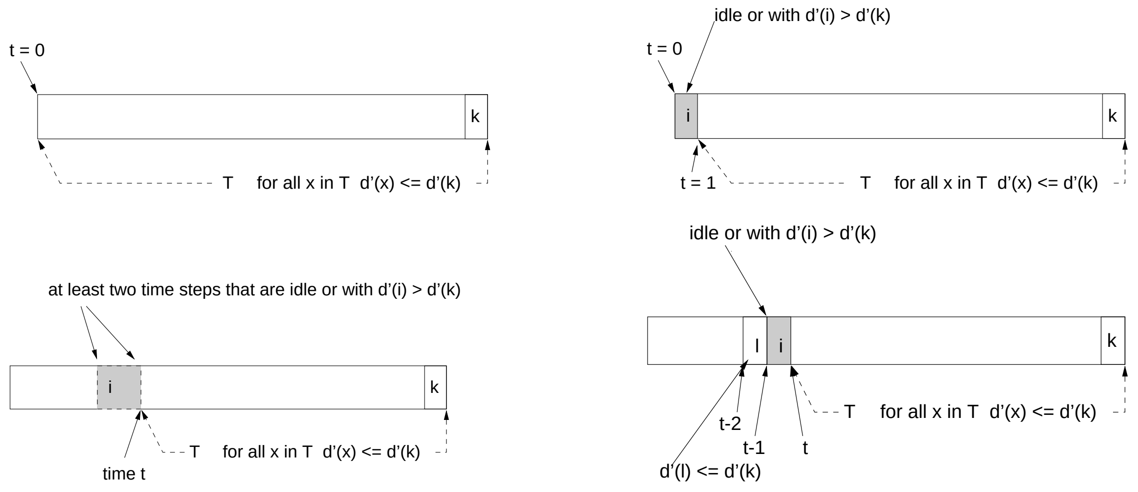

t = 0
k
T for all x in T d'(x) <= d'(k)
idle or with d'(i) > d'(k)
t = 0
i
k
t = 1
T for all x in T d'(x) <= d'(k)
at least two time steps that are idle or with d'(i) > d'(k)
i
k
time t
T for all x in T d'(x) <= d'(k)
idle or with d'(i) > d'(k)
l
i
k
t-2
t-1
t
d'(l) <= d'(k)
T for all x in T d'(x) <= d'(k)

Figure 10: The four cases in the proof of theorem 4: instruction k is the first to miss the deadline and free time slots are shaded.

at least one of $I_i \in T$ will miss its modified deadline in all schedules. This case also contradicts the assumption.

case 3 — There exists a bound segment T from time t to $\sigma(I_k)$. Furthermore, time steps $t-1$ and $t-2$ are free. By greediness this implies for all $I_i \in T$, $s_i \geq t$. But then by a pigeon hole argument at least one of $I_i \in T$ will miss the deadline.

case 4 — There exists a bound segment T from time t to $\sigma(I_k)$. Furthermore, time step $t-1$ is free but time step $t-2$ is bound. By greediness this implies that for all $I_i \in T$, $(I_l \sqsubset I_i \wedge w(l,i) \geq 1) \vee s_i \geq t$. By the definition of the modified deadline and lemma 2 we know that either (1) one of $I_i \in T$ must miss the deadline in all schedules, or (2) I_l must miss the deadline, thus contradicting that I_k is the first to miss the deadline.

This completes the proof. ■

Our algorithm also yields a similar theorem when applied to the problem of precedence constrained 2-processor scheduling with individual start-times and deadlines[18]. The proof of this result is similar and is omitted.

We can invoke the previous algorithm in a binary search mode to locate either the minimal makespan and/or the minimal tardiness schedule, which takes at most $O(\log n)$ trials. Let s_{max} denote the maximal start-time, then any feasible schedule must have makespan between $s_{max}+1$ and $s_{max}+2n$. Thus we have the following corollary:

Corollary 5 *For the restricted instances of one-pipeline and (0,1)-latencies, the minimal makespan schedule can be computed with $O(\log n)$ invocation of the rank algorithm. Similarly, the minimal tardiness schedule can be computed in the same time bound.*

3.4.1 Running Time

We have already shown the correctness of the rank algorithm and its other properties. Let us now analyze its running time.

Theorem 6 *Algorithm 1 is correct and terminates with stabilized modified deadlines or with "infeasible schedule" as output in time $O(n^2 \log n)$ for an arbitrary number of pipelines, latencies, and type-1 constraints, when the input is given in transitively closed form.*

Proof: Step (1) of the algorithm takes time $O(E' \log n)$ as shown in [40], where E' is the number of data-dependence edges in the transitively closed DAG[11]. Sorting the modified deadlines and the start-times in steps (2) and (3) can be performed in time $O(n \log n)$. Backward scheduling with the successors in step (4) takes a total time of $O(E' \log n)$. In each iteration of step (4) the set $indep_i$ can be added to the initial successors backward schedule in time $O(n \log n)$. Finally, step (5) just reiterates the algorithm once so it does not alter the complexity. The time is bounded by the iteration in step (4), and is thus $O(E' \log n + n^2 \log n)$ or $O(n^2 \log n)$. ■

3.4.2 The Algorithm for Time_tract Specifications

Utilizing the above algorithm as a "black-box," it is now possible to reduce type-2 constraints described in section 2.2.1 to type-1 constraints. In fact, w.l.o.g. we'll consider general (monotone) constraints of the form $s_i \geq \alpha \cdot \mathbf{s} + \mathbf{a}$ and $d_i \geq \beta \cdot \mathbf{d} + \mathbf{b}$ where $\alpha, \beta \in \mathbf{Q}_+^n$, $\mathbf{a}, \mathbf{b} \in \mathbf{Q}$ with the restriction that $\alpha_i = 0$ and $\beta_i = 0$[12].

Partition the constraints into start-times and deadlines inequalities. Observe that the minimal solution for $\mathbf{s}$ and the maximal solution for $\mathbf{d}$ in $\mathbf{N}^n$ are well-defined if they exist. For the general monotone constraints, the optimization problem can be readily formulated as a linear programming problem and solved using linear programming techniques. For the type-2 constraints of interest to us, the decision question is reducible to computing single-source shortest paths. Given Γ inequalities on n variables this problem can be solved efficiently using the Bellman-Ford algorithm in time $O(\Gamma n)$.

4 Global Scheduling Issues

We now refocus our attention on global scheduling.

As is standard in conventional global scheduling [3, 10, 15, 46], we treat separately loop scheduling [46, 32] and acyclic program regions scheduling [3, 10, 15]. We advocate taking this approach also for the case of scheduling programs instrumented with Time_tract specifications. In this extended abstract, we only discuss key issues in scheduling acyclic program regions; related expressive power questions will be discussed in Section 6. Details of loops will be deferred to the full paper.

The input to an acyclic global scheduler is very similar to the that for basic-blocks defined in Section 3.1 with a few essential extensions. First, because of branching, we must also be

[11]Please note that transitive closure is a standard preprocessing step and the E' term plays a role in the running times of conventional instruction scheduling algorithms as well [41].

[12]$\mathbf{Q}_+$ is the set of non-negative rationals.

sensitive to the program's control-flow in addition to data-dependence. In particular, moving instructions out of their initial basic-blocks past program *branch* and *merge* points [1] in the CFG lead to "side-effects" which require special attention and add to the complexities.

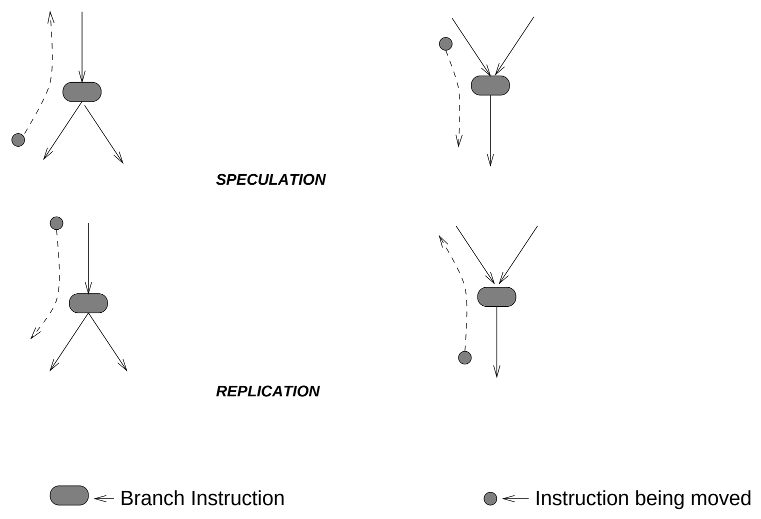


Figure 11: Speculation and replication.

A significant issue that arises in the context of moving instructions globally is the effects of *speculation* and *replication*. In either case these movements may require expensive repair, leading to *overheads*, as detailed in Figure 11; for a complete discussion of these issues, please see [15].

In this figure, the top row details code motion where the instruction goes from being executed *sometimes* since it is on one of the two conditional branches, to being executed *always* after the code-motion. This type of an execution is referred to as speculative. The complementary situation is shown in the two figures in the bottom row. In this case, after code motion, instruction I is executed only when the left branch is taken, whereas previously, it is executed independent of the branch. The solution to preserving the semantics of the program in this case is to make a copy of I on the right branch as well — hence, the term replication.

Both of these side-effects can lead to extra costs and consequently the designers limit the degree to which instructions can cross basic-block boundaries.

In our global scheduling model, we will permit instructions that are not branches or merges to be moved without any restrictions. In contrast, branches (and merges) are treated differently since the corresponding overheads can be prohibitive [16]. Consequently, current approaches to scheduling is to limit the relative reordering of these instructions completely. We will retain this restriction in our subsequent discussion.

Assume an acyclic (possibly branching) program P with instruction set $\mathbf{I}$. Such a program can be represented as an acyclic directed graph (the control flow graph) in which each node is labeled by a single instruction. Branching instructions label nodes with two departing edges, while merge node have two entering edges. The *start* node has a single successor but no predecessor, while the *stop* node may have several predecessors (if it is also a merge node) but no successors. All other nodes have each a single successor and a single predecessor. We refer to the branching, merge, start, and stop nodes as the *pivotal nodes* in the program.

A *schedule* for program P is another DAG S such that there is a 1-1 type-preserving correspondence between the pivotal nodes of P and the pivotal nodes of S. For simplicity, we assume that they are called by the same names (or same indices) in both graphs. Type preservation means that a node n has the same number of successors and predecessors in P as it has in S. Nodes in a schedule may be labeled by sets of instructions, implying that all the instructions labeling a node are initiated at the same execution cycle. The instructions of S are partitioned into $\hat{\mathbf{I}} = \tilde{\mathbf{I}} \cup \mathbf{I}'$, where $\tilde{\mathbf{I}}$ are duplicates of the instructions of P, while $\mathbf{I}'$ are *repair instructions* intended to undo the effect of some preceding instructions. The need for more than one copy of an original P-instruction and for repair instructions rises due to speculative code motion, which is allowed in our model. It is required that the instruction labeling a *start* or a branching node n in P is included in the label of n in S, and that these instructions have precisely one copy in $\tilde{\mathbf{I}}$. We also disallow including in a single S-label two instructions which assign values to the same variable.

For a given set of inputs, we can simulate the execution of S by performing at each cycle all the instructions labeling the current control node, and branching whenever we reach a branch node. In such a simulation, we should take care that the execution of an instruction $\mathbf{I}$ with latency 1 should have its effect (e.g., a value assigned to a variable) with the appropriate delay of one cycle. A schedule S is considered to be *feasible schedule* of program P if

1. Each label of S contains no more than num_l instructions of type l.
2. For each set of inputs, the final state reached by program P is identical to the final state reached by simulation of S on these inputs.
3. For each original instruction $I \in \mathbf{I}$, no simulation run of S should execute more than a single copy of I.

For the case that timing constraints are associated with program P, we say that the P-feasible schedule S *respects the timing constraints* if it also satisfies

4. For each timing constraint $\sigma(I_j) - \sigma(I_i) \sim k_{ij}$ (where $\sim$ is $\leq$ or $\geq$), and each simulation run of S in which some copy of instruction I_i was executed at cycle τ_i while some copy of instruction I_j was executed at cycle τ_j, it is required that

$$\tau_j - \tau_i \sim k_{ij}.$$

5 ET_Tract and Notation

Starting with the case of basic-blocks, we consider permitting constraints on the time differences between the execution of instructions associated with markers (relative time constraints). Given instructions I_i and I_j, we allow constraints of the form $(\sigma(I_j)) - (\sigma(I_i)) \leq c_{j,i}$. We refer to this extended specification language by the name of ET_tract. We note that this small extension of the language causes the problem of determining the existence of a feasible schedule to become NP-complete even for the following very restricted family of constraints, basic-blocks and processor.

Theorem 7 *Finding feasible schedules is NP-complete for ET_tract specifications even for the case of a single non-pipelined processor ($k = 1$), basic-blocks which have no data-dependences, and only relative time constraints of the form $\sigma(I_i) - \sigma(I_j) \leq c_{i,j}$, where all non-zero values of the constant $c_{i,j}$ are equal to one another.*

We note that in contrast, with arbitrary Time_tract specifications, we have a fast polynomial time algorithm for instruction scheduling detailed before, which can cope with (i) arbitrary data-dependences, (ii) two-stage pipelines (Theorem 4).

5.1 Dynamic Markers and Global scheduling

Let us further consider global scheduling as described before for acyclic regions. In particular, we wish to permit the possibility of a marker in this context to denote an event which occurs when an instruction is executed while a certain data-dependent predicate holds. For instructions I_i, I_j, and predicates p_i, p_j, we allow constraints of the form $\sigma(I_i \wedge p_i) - \sigma(I_j \wedge p_j) \leq c_{i,j}$. We refer to the conjunction $I_i \wedge p_i$ as a *dynamic marker*. In ET_Tract, we permit predicates which, given a global schedule for the acyclic program (region), can be verified in NP. So, for example, p_i and p_j can be arbitrary propositional formulas, or a system of integer linear constraints on the program variables. Of course, the programmer can involve auxiliary predicates which are not part of the program itself as parts of a dynamic marker.

Comment about NP membership: We advocate limiting the power of predicates to those that can be determined in NP since classical global scheduling problems—most of which are NP-complete—have "industrial strength" heuristics. Therefore, membership in NP is preliminary theoretical evidence of being amenable to such techniques and hence tractable in a pragmatic sense.

6 Expressive Power and Timed-automata

To compare the approach presented here with other time analysis approaches, we show how to formulate the scheduling problems we consider in this paper in terms of timed automata [4]. The presentation of timed automata adopted here is a variation on the model of guarded-command real-time programs presented in [13].

We assume a finite set of *system variables* V which can be partitioned into $V = D \cup C$, where $D = \{x_1, \ldots, x_m\}$ is the set of *discrete control variables*, ranging over the finite domain $\mathcal{D} = \{0, \ldots, d\}$ and $C = \{t_1, \ldots, t_n\}$ is a set of *clocks*, ranging over the natural numbers $\mathbf{N}$. A *state* s is an interpretation of all system variables by values of their corresponding types, i.e., $s[x_i] \in \{\mathrm{T}, \mathrm{F}\}$ and $s[t_j] \in \mathbf{N}$, for every $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. We denote by $\mathbf{N}^+$ the set of all positive integers. For a state s and delay δ, we denote by $s + \delta$ the state s' obtained by adding δ to all clocks. That is, $s'[x_i] = s[x_i]$ and $s'[t_j] = s[t_j] + \delta$, for every $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$.

The sets of *state predicates* ϕ and *$\mathcal{D}$-expressions* E are defined inductively by the grammars

$$\begin{array}{lclclclclclcl}
\phi & ::= & x_i = c & | & t_j \leq a & | & a \leq t_j & | & t_j + a \leq t_k + b & | & \neg\phi & | & \phi_1 \wedge \phi_2 \\
E & ::= & x_i & | & c & | & \multicolumn{3}{c}{\textbf{if } \phi \textbf{ then } E \textbf{ else } E}
\end{array}$$

for discrete variable $x_i \in D$, clocks $t_i, t_j \in C$, and natural constants $c \in \mathcal{D}$ and $a, b \in \mathbf{N}$. State predicates and $\mathcal{D}$-expressions are evaluated over states in the usual way, and we denote by $s[\phi]$ and $s[E]$ the result of such evaluation. We write $s \models \phi$ to indicate that $s[\phi] = \mathrm{T}$.

A (*guarded command*) *real-time* program (RTP) $\mathcal{P} = (V, \Theta, G, \phi^\square)$ consists of

- $V = D \cup C$ – the set of *system variables*.
- Θ – the *initial condition*, a state predicate characterizing all the initial states of the program. It is required that $\Theta \rightarrow t_j = 0$, for every $j \in \{1, \ldots, n\}$.
- G – the *program body*. a set of guarded commands. Each *guarded command* is of the form
$$\psi \quad \rightarrow \quad (x_1, \ldots, x_m, t_1, \ldots, t_n \ := \ E_1, \ldots, E_m, f_1, \ldots, f_n), \qquad (1)$$
where ψ is a state predicate (the guard of the command), followed by a multiple assignment assigning new values to all system variables, such that f_j is either t_j or 0. Thus, a clock that is not reset to 0 retains its previous value. Assignments of the forms $x_i := x_i$ or $t_j := t_j$ are omitted from the presentation of concrete RTP's.
- $\phi^\square$ – the *program invariant*, identifying a condition which all program states should satisfy. We require that $\phi^\square$ be *past-closed*; that is, for all states s and delay $\delta \in \mathbf{N}^+$, $s + \delta \models \phi^\square$ implies $s \models \phi^\square$.

For $\delta \in \mathbf{N}$, we say that the state s' is a *δ-successor* of state s, denoted by $s \xrightarrow{\delta}_{\mathcal{P}} s'$, in one of two cases:

(1) either $\delta > 0$, and $s' = s + \delta$ satisfies $\phi^\square$, or

(2) $\delta = 0$, and there exists a guarded command of the form (1), such that $s \models \psi$, $s'[x_i] = s[E_i]$ and $s'[t_j] = s[f_j]$, for every $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, where $s[0] = 0$.

A *computation* of an RTP $\mathcal{P}$ is an infinite sequence of states and nonnegative delays

$$\kappa = s_0 \xrightarrow{\delta_0}_{\mathcal{P}} s_1 \xrightarrow{\delta_1}_{\mathcal{P}} s_2 \xrightarrow{\delta_2}_{\mathcal{P}} s_3 \xrightarrow{\delta_3}_{\mathcal{P}} \cdots,$$

starting with an initial state ($s_0 \models \Theta$), and proceeding by legal $\mathcal{P}$-steps ($s_i \xrightarrow{\delta_i}_{\mathcal{P}} s_{i+1}$, for every $i = 0, 1, \ldots$).

The Reachability Problem for RTP's

A *reachability problem* consists of an RTP $\mathcal{P}$ and a state predicate ϕ over the system variables of $\mathcal{P}$. The problem is to decide whether there exists a $\mathcal{P}$-computation $\kappa : s_0, s_1, \ldots$ and a position $p \in \mathbf{N}$ such that $s_p \models \phi$. As can be inferred from the analysis presented in [4], this problem is PSPACE-complete.

Encoding Instruction Scheduling Problems as RTP's

We will show that any instruction scheduling problem (ISP) can be encoded as an RTP reachability problem. Here we restrict our attention to the case of basic blocks scheduling and M functional units, all of the same type.

We assume that an ISP is presented by a dependency graph $(\mathbf{I}, E)$, where $\mathbf{I} = \{I_1, \ldots, I_q\}$ are the program's instructions, and $E \subseteq \mathbf{I} \times \mathbf{I}$ is a set of edges, denoting the dependencies between the instructions. The latency information is presented by the latency function w. Also given is a parameter M, specifying the number of available functional units.

The timing constraints are partitioned into the set of *lower-bounds* $\mathcal{L}$ and the set of *upper-bounds* $\mathcal{U}$. Each lower-bound constraint has the form $\sigma_j - \sigma_i \geq L_{ij}$, where σ_i and σ_j are the initiation times for the execution of instructions I_i and I_j, respectively, and $L_{ij} \in \mathbf{N}$. An upper-bound constraint has the form $\sigma_j - \sigma_i \leq U_{ij}$, where $U_{ij} \in \mathbf{N}$.

The dependency conditions can be integrated into the lower-bounds constraints by defining an *extended lower-bound* set $\mathcal{L}^+$ which consists of the original lower-bound constraints $\mathcal{L}$, to which we add for each dependency $(I_i, I_j) \in E$ the constraint $\sigma_j - \sigma_i \geq 1 + w(I_i, I_j)$.

Given an ISP $\Pi = (\mathbf{I}, E, w, \mathcal{L}, \mathcal{U}, M)$, we will show how to encode it as an RTP reachability problem $(\mathcal{P}_\Pi, \phi_\Pi)$.

As system variables we take $V_\Pi = \underbrace{u, x_1, \ldots, x_q}_{D} \cup \underbrace{t_0, t_1, \ldots, t_q}_{C}$, consisting of the *functional unit counter* u, *instruction status* variables $x_1, \ldots, x_q$, indicating the initiation status of the instructions, and *instruction clocks* $t_1, \ldots, t_q$, measuring the time since the initiation of the corresponding instruction. The clock t_0 measures the time from the beginning of the execution of the program. The variable u ranges over the domain $\{0, \ldots, M\}$, while $x_1, \ldots, x_q$ range over $\{0, 1, 2\}$.

The initial condition is given by Θ_Π: $u = M \wedge x_1 = \cdots = x_q = 0 \wedge t_0 = t_1 = \cdots = t_q = 0$.

The body: With each instruction I_j, $j = 1, \ldots, q$, we associate the following commands:

$$\begin{array}{lrcl}
g_j^0: & u > 0 \wedge x_j = 0 \wedge \bigwedge\limits_{(\sigma_j - \sigma_i \geq L_{ij}) \in \mathcal{L}^+} (x_i > 0 \wedge t_i \geq L_{ij}) & \rightarrow & (u, x_j, t_j := u - 1, 1, 0) \\
g_j^1: & x_j = 1 \wedge t_j = 1 & \rightarrow & (u, x_j, t_j := u + 1, 2, t_j)
\end{array}$$

Command g_j^0 initiates the execution of instruction I_j. It is enabled only when at least one functional unit is available ($u > 0$), I_j has not been previously initiated ($x_j = 0$), and no (extended) lower-bound constraint of the form $\sigma_j - \sigma_i \geq L_{ij}$ would be violated if we initiate I_j now. The command subtracts 1 from the available-units counter, sets the status variable x_j to 1, and resets the clock t_j. Command g_j^1 releases of the functional unit which has been executing I_j.

The program invariant: For each upper-bound constraint $(\sigma_j - \sigma_i \leq U_{ij}) \in \mathcal{U}$, invariant $\phi_\Pi^\Box$ contains the conjunct $x_i > 0 \wedge x_j = 0 \quad \rightarrow \quad t_i \leq U_{ij}$, implying that the value of t_i cannot exceed U_{ij} if instruction I_i has been initiated while I_j has not. This forces I_j to be initiated no later than U_{ij} time units after I_i is initiated.

In addition, $\phi_\Pi^\Box$ contains, for each instruction I_j, the conjunct $x_j = 1 \quad \rightarrow \quad t_j \leq 1$. This conjunct requires the release of the functional unit executing I_j no later than one time unit after its initiation. This shows that the guarded commands take care of the lower bounds, while the program invariant takes care of the upper bounds.

The goal predicate is given by ϕ_Π: $x_1 = \cdots = x_q = 2$. This predicate describes a state in which all instructions have completed their execution. If such a state is reached it implies that we managed to find a computation which satisfies all the originally specified real-time constraints.

Complexity of the Translated Problem

As previously stated, the complexity of the general reachability problem for RTP's is PSPACE-complete. However, the RTP we get by encoding an instruction scheduling problem has a special structure, which leads to a better complexity.

An RTP $\mathcal{P}$ is called *acyclic* if, in every computation of $\mathcal{P}$, every guarded command can be executed at most once. It is not difficult to see that the translation of an an instruction scheduling problem (for an acyclic dependency graph) yields an acyclic RTP. The reachability problem for acyclic RTP's is considerably simpler than for the general case, as is stated by the following claim.

Claim 1 *The reachability problem for acyclic* RTP*'s is* NP*-complete.*

If we restrict our attention to ISP's with Time_tract constraints and 0-1 latency, this leads to a still simpler version of RTP's. Namely, this corresponds to the case that the only references to clocks in the state predicates are of the forms $t_0 \leq a$, $t_0 \geq b$, or $t_j = 1$, for an arbitrary $j \neq 0$. For this restricted type of RTP's (which we call *Time_tract rtp's*), we have the following stronger claim:

Claim 2 *The reachability problem for a* Time_tract-RTP *can be solved in time* $O(n^2 \log n)$.

Extensions of the Reduction

Let us consider the extensions that are necessary in order to handle the more general cases. Extending the translation to handle functional units of different types is straightforward. Dealing with possibly branching acyclic regions is more intricate. For this more complex case, we no longer use a reduction into an RTP-reachabilty problem. Instead, we formulate a new *realizabilty problem* for RTP's, and show that the scheduling problem for acyclic regions with ET_tract specifications can be reduced to the realizabilty problem for RTP's. The situation here is similar to the one described in Claim 1 (without the specially efficient case discussed in Claim 2). Namely, the general realizability problem is PSPACE-complete. However, restricted to acyclic RTP's it becomes "only" NP-complete. This will be discussed in greater detail in the full paper.

7 Comparison with Other Work and Remarks

We are the first to present a fast (polynomial time) algorithm in the context of RISC pipelines with 0/1 latencies that can provably solve the scheduling problem given Time_tract specifications; for the same specifications, the algorithm in [18] can solve the problem for the simpler case of two independent and identical processors; chain data-dependence structures are known to be schedulable in $O(n^2)$ time [24]. Given a program with ET_Tract constraints, linear task dependences but the task execution times are known only in terms of upper- and lower-bounds—this is the case for coarse granularity tasks at the OS level for example—a scheduling algorithm called *parametric dispatching* is known [19][13][14]. We are the first to formally characterize and analyze the expressive power of Time_tract and ET_Tract (Sections5 and 6), which are specification languages whose decision questions are in NP. A notation for specifying time called *TCEL* and a framework for scheduling programs derived from TCEL specifications were introduced in [22]. Contrasting with our approach and results here, TCEL is a programming language wherein timing constraints are explicitly specified as part of control-flow of the application program; in this sense, the programmer must write TCEL programs. Furthermore, this approach does not specify an algorithm with provable properties, nor a formal model and characterization of the expressive power of TCEL; additional heuristic proposals for scheduling tasks with ET_Tract constraints have also been made [49]. In the context of basic-blocks, we have verified that, in terms of expressibility, Time_tract $\subset$ ET_Tract $\equiv$ TCEL. For acyclic program regions, ET_Tract $\subset ?$ TCEL $\subset$ RTP (defined in Section 6). We conjecture that TCEL encodes more complex decision questions than ET_Tract for acyclic program regions, but as denoted by $\subset ?$, this inclusion is open at the time of writing this paper. The inclusions are based on the standard separations holding between P, NP, Co-NP and PSPACE.

[13]For more general constraints, this paper proposes an exponential time algorithm based on Fourier-Motzkin elimination

[14]Our comparison is restricted to work on instruction scheduling which has to be contrasted with classical real-time process scheduling [34] and the related large body of work including interactions between the compiler and the OS schedulers [23].

References

[1] *Compiler Construction* A. Aho, R. Sethi and J. Ullman, Addison-Wessley, 1984.

[2] *Reference Manual for the Ada Programming Language* ANSI/MIL-STD-1815A-1983.

[3] A. Aiken and A. Nicolau. Loop quantization; An analysis and algorithm. Tech report 87-821, Cornell University, March, 1987.

[4] R. Alur and D. Dill. A Theory of Timed Automata. In *Theoretical Computer Science*, vol. 126, 183–235, 1994.

[5] R. Alur and T. Henzinger. Logics and Models of Real-time: A Survey. In *Real-time Theory in Practice*, Lecture Notes in Computer Science, 600, Springer-Verlag, 1991.

[6] E. Asarin, O. Maler and A. Pnueli. Symbolic Controller Synthesis for Discrete and Timed Systems. In *Hybrid System II*, Lecture Notes in Computer Science, 999, Springer-Verlag, 1995.

[7] T. Ball and J. Larus. Branch Prediction for Free. *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation* June 1993.

[8] R. E. Barr, J. Bashyam, D. Messenger, P. Norwood and K. Palem Multichannel Real-Time Analysis of the Clinical EEG on a Dual Microprocessor System. *J. Clinical Engineering,* Vol. 9, 1984.

[9] D. Bernstein and I. Gertner. Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle. In *ACM Transactions on Programming Languages and Systems*, 11(1), 57–66, 1989.

[10] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* 1991.

[11] Pradeep. Dubey, Kevin. O'Brien, Kathryn. O'Brien and Charles Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-grained Multithreading. preprint, 1995.

[12] S. Moon and K. Ebciougulu. An Efficient Resource-constrainted Global Scheduling Technique for Superscalar and VLIW Processors. *Proceedings IEEE MICRO-25*, 1992.

[13] T. Henzinger, X. Nicollin, J. Sifakis and S. Yovine. Symbolic Model-Checking for Real-Time Systems. In *Inform. and Comput.*, vol. 111, 193–244, 1994.

[14] J. Ferrante K. J. Ottenstein and J. D. Warren. The Program Dependence Graph and its use in Optimizations, *ACM Transaction on Programming Languages and Systems*, vol. 9, 319–349, 1987.

[15] J. Fisher. Trace Scheduling: A General Technique for Global Microcode Compaction. *IEEE Transactions on Computers,* C-30(7):478–490, 1981.

[16] Joseph Fisher. Global Code Generation for Instruction-level Parallelism: Trace Scheduling-2 1991

[17] M. Garey and D. Johnson Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1979.

[18] M. Garey and D. Johnson Two-processor Scheduling with Start-times and Deadlines. *SIAM J. Computing,* vol. 6: 416–426, 1977.

[19] R. Gerber, W. Pugh and M. Saksena Parametric Dispatching of Hard Real-time Tasks *IEEE Transactions on Computers,* vol. 44: 1995.

[20] S. Freudenberger, T. Gross, P. Lowney. Avoidance and Suppression of Compensation Code in a Trace Scheduling Compiler. *ACM Transactions on Programming Languages and Systems*, vol. 16, 1156–1214, July 1994.

[21] P. Gibbons and S. Muchnick. Efficient Instruction Scheduling for Pipelined Architecture. *Proceedings of the ACM Symposium on Compiler Construction*, 11–16, 1986.

[22] S. Hong and R. Gerber. Compiling Real-Time Programs with Timing Constraint Refinement and Structural Code Motion. *IEEE Transactions on Software Engineering*, vol. 21, May 1995; preliminary version appeared in Compiling Real-Time Programs into Schedulable Code. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation,* 1993.

[23] S. Hong and R. Gerber. Scheduling with Compiler Transformations: the TCEL Approach. *Proc. IEEE Workshop on Real-time Operating Systems and Software*, May 1993.

[24] C.Han and K. LIn. *Job Scheduling with Temporal Distance Constraints*, TR-UIUCDCS-R-89-1560, Univeristy of Illinois, 1989.

[25] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* vol. 8, 1987.

[26] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM TOPLAS,* 5(3), 1983.

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

[28] J. Hennessy and D. Paterson. Computer Architecture: A Quantitative Approach *Morgan Kaufmann*, 1990.

[29] W.-M. W. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. The Journal of Supercomputing, Vol. 7 (1993), 229-248.

[30] M. Katavenis. Reduced Instruction Set Architecture for VLSI. *MIT Press*, Cambridge MA, 1984.

[31] V. Kathail, M. Schlansker and B. R. Rau. HPL PlayDoh Architecture Specification Version 1.0 *HPL-93-80*, HP Labs, Palo Alto, CA, February 1994.

[32] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proceedings SIGPLAN'88 Symposium on Programming Language Design and Implementation*, 318–328, 1988.

[33] M. Smith, M. Horowitz and M. Lam. Efficient Superscalar Performance Through Boosting. *Proceedings Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,* 248–261, 1992.

[34] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real Time Environment. *J. ACM*, vol. 20 (1), 1973.

[35] O. Maler, A. Pnueli and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In *Proc. of STACS'95*, Lecture Notes in Computer Science, 900, Springer-Verlag, 229–242, 1995.

[36] Z. Manna and R. Waldinger, Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, Vol. 18, 674–704, 1992.

[37] N. Nachiappan. Personal Communications and Memorandum of Support. 1995.

[38] K. Palem On the Complexity of Precedence Constrained Scheduling. *TR-86-11*, University of Texas, Austin, TX, 1986.

[39] K. Palem and V. Sarkar. Code Optimization in Modern Compilers. *Western Institute of Computer Science*, Stanford University, CA, 1995.

[40] K. Palem and B. Simons. Scheduling Time-critical Instructions on RISC Machines. *ACM TOPLAS*, 5(3), 1993.

[41] K. Palem and B. Simons. Instruction Scheduling. In *Optimization in Compilers,* (eds: F. Allen, B. Rosen and K. Zadeck). ACM Press and Addison-Wesley (to appear).

[42] D. Paterson. Reduced Instruction Set Computers. *Communications of the ACM,* 28(1):8–21, 1985.

[43] D. Paterson. T. Anderson, D. Culler and D. Patterson, A Case for NOW (Networks of Workstations. *IEEE Micro*, vol. 1995.

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

[45] *Proceedings of the ACM Sigplan Workshop on Languages, Compilers and Tools for Real-time Systems,* La Jolla, California, June 1995.

[46] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Proceedings of the 27th Annual Symposium on Microarchitecture*, December 1994.

[47] B. Rau and J. Fisher. Instruction Level Parallel Processing: History, Overview and Perspective. *J. Supercomputing*, vol.7, 9—50, 1993.

[48] R. Russell and R. Grewell. Software Aids Pull for Real-time RISC: RISC/CISC Tradeoffs. *Electronic Engineering Times*, September 1994, Vol. 51.

[49] M. Saksena, R. Gerber and A. Agrawala Scheduling with Relative Timing Constraints. *IEEE Workshop on Real-time Operating Systems and Software*, May 1993.

[50] D. Wall. Predicting Program Behavior Using Real or Estimated Profile. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 59–70, 1991.

[51] Venix Real-time Programmers Manual *Venturcom*, 1989.

[52] H. Warren. Instruction Scheduling for the IBM RISC System/6K Processors. *IBM Journal of Research and Development,* 85–92, 1990.