Single-mode, single-processor Giotto scheduling

Benjamin Horowitz
bhorowit@cs.berkeley.edu

Report No. UCB/CSD-03-1238

April 16, 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Abstract

This report presents a new algorithm for scheduling single-mode Giotto programs for a single processor. We extend the classical scheduling problem $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$ to an infinite, periodic variant. We present a polynomial time algorithm for this variant that finds a feasible schedule whenever one exists. We show how to embed the problem of scheduling a class of single-mode Giotto programs for a single processor into our more general problem. The embedding yields a pseudopolynomial-time algorithm that allows more programs to be scheduled than previous techniques. Finally, we present a technique to aggregate distinct activities into the same thread, and we show how to execute Giotto programs using a single stack.

1 Introduction

In this report, we address the problem of scheduling single-mode Giotto programs for a single processor. This is an important case, as many control algorithms have a single mode of operation, and many control system implementations use a single processor to reduce cost, design time, and debugging effort. Instead of developing scheduling algorithms for Giotto only, we adopt a more inclusive approach. Our strategy is to embed the single-processor scheduling problem for single-mode Giotto programs into a more general problem. It is this more general problem which we solve. Our hope is that the scheduling techniques that we develop will prove useful not only for Giotto, but also for more expressive time-triggered programming languages, e.g., languages in which the unit-delay requirements of Giotto are relaxed.

Our approach has two ingredients. The first is our use of precedence constraints to model data flow. An execution of a Giotto program requires running a set of *jobs* on a CPU; if a job j writes a value to a port that another job j' reads, then j must complete before j' begins. Besides sensors and actuators, all jobs may be executed at any time, subject to the precedence constraints imposed by data flow. This precedence-constrained view of single-processor scheduling lets more Giotto programs be scheduled than was previously possible. The second ingredient is our use of the scheduling algorithm EDF with precedence constraints [Bła76], which we term EDF$^{\prec}$. This algorithm provides a useful departure point, but to make it appropriate for scheduling Giotto, we extend it to handle an infinite, periodic set of jobs. The algorithm that we develop has two desirable properties. First, it is *optimal*: it finds a schedule satisfying timing and precedence constraints whenever such a schedule exists. Second, it is reasonably fast: it runs in time polynomial in its input size, and pseudopolynomial in the frequencies of the Giotto program. Though precedence-constrained *multiprocessor* scheduling of programming languages is an active area of research [DRV00], to the author's knowledge precedence-constrained single-processor scheduling of programming languages has not been extensively studied. The results of this report are not likely to generalize to a multiprocessor setting: in such a context, our models bear a strong similarity to job shop scheduling, which is notoriously difficult, both practically and theoretically [Pin95].

An EDF-based schedulability test was previously presented in [HKMM02]. Precedence constraints play no part in this algorithm. This earlier algorithm optimally schedules Giotto programs for a single processor, *under three restrictions*:

1. Each task driver executes at the time instant τ_i specified in the Giotto semantics.
2. Each task invocation executes in the interval $[\tau_i, \tau_i + \pi/\omega]$, where π is the period of the mode invoking the task, and ω is the invocation's frequency.
3. Sensors, actuators, and task drivers require "negligible" computation time. Exactly how much computation time is negligible was not specified.

It was subsequently argued in [HHK03] that, from a semantic perspective, these restrictions can be relaxed in a systematic way. In this report, we continue the line of argument of [HHK03], showing how the restrictions can be relaxed from a *scheduling* perspective as well. It should be noted that though our algorithm allows more programs to be scheduled, it runs in time pseudopolynomial in the frequencies of the input program, whereas the test of [HKMM02] is polynomial-time. This disadvantage seems rather slight, as the frequencies of a Giotto program are typically small.

For simplicity, the algorithm that we develop is a *pre-runtime* scheduling algorithm: before runtime, it produces a complete schedule of the implementation's threads. This schedule specifies when to start, suspend, resume, and stop each thread. A pre-runtime scheduling algorithm has two advantages. First, it minimizes the complexity of the actions at runtime. The runtime "scheduler" becomes highly deterministic, which greatly simplifies debugging. Second, the generated schedules can be independently verified prior to runtime. This provides an important double-check in situations where safety and reliability are primary concerns. The advantages of pre-runtime scheduling are thoroughly discussed in [Kop97]. Of course, other approaches are also possible. For example, one might provide a schedulability *test* prior to runtime, but relegate all *decisions* about processor allocation to a runtime scheduler. The first use of this approach is perhaps [LL73]; a recent exemplar is [BHR93]. Additionally, one might make the runtime scheduler more clever about how to handle situations of overload (see, e.g., [BS93]). Neither of these approaches is inconsistent with the approach pursued here; we have adopted a pre-runtime approach only in order to study scheduling models and algorithms in as simple a setting as possible.

The structure of this report is as follows. Section 2 presents two examples which motivate the need for precedence-constrained scheduling. These example cannot be scheduled by the current Giotto compiler, but can be scheduled by the algorithm in this report. Section 3 discusses models and algorithms for precedence-constrained scheduling problems. We extend the model $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$ to an infinite, periodic variant. We then present an optimal, polynomial-time scheduling algorithm for this variant. Section 4 shows how to translate single-mode Giotto programs into instances of the model developed in Section 3. Section 5 describes two additional optimizations, the aggregation of multiple activities into fewer operating system threads, and the use of a single stack to execute activities.

A word on notation: the symbol $\mathbb{Z}$ denotes the integers, and $\mathbb{R}$ denotes the reals. For an ordered set S and an element $s \in S$, $S^{\geq s}$ (respectively, $S^{>s}$) denotes the set $\{s' \in S \mid s' \geq s\}$ (respectively, $\{s' \in S \mid s' > s\}$). Finally, the symbol $[\ell \mathinner{..} u]$ denotes the set $\{i \in \mathbb{Z} \mid \ell \leq i \leq u\}$.

2 The need for precedence-constrained scheduling

In this section, we motivate the need for precedence-constrained single-processor scheduling of single-mode Giotto programs. We present several examples that the current Giotto compiler is not able to schedule, but that are nonetheless schedulable. The first example, in Section 2.1, indicates that all jobs of a Giotto program should be preemptible, not just task invocation jobs. The second example, in Section 2.2, argues that the execution of jobs of one round of a Giotto program should be allowed to continue into the next round of the program. These examples are two among many; we have included these particular examples in order to argue that the scheduling requirements of the current compiler are overly restrictive (e.g., the requirement that drivers execute "synchronously," and that tasks finish before their "logical" deadlines; cf. [KSHP02] and [HKMM02]). We shall use the example of Section 2.2 as a running example in Section 3.

2.1 Preemptible drivers

Figure 1 shows a Giotto program. This program has two sensors s_1 and s_2, each taking 1 unit of time to read.[1] There are two tasks t_1 and t_2, and their respective drivers d_1 and d_2. Each of these takes 1 unit of time to execute, except driver d_2, which takes 2 units. A third driver d_3 writes actuator a and takes 1 unit. There is a single mode m with period 12. Mode m invokes t_1 with frequency 2, t_2 with frequency 1, and d_3 with frequency 2.

We now describe some of the timing requirements of the program of Figure 1. Figure 2 depicts these requirements in graphical form. Thick blue boxes indicate jobs that execute at a fixed time. Thin black boxes indicate jobs that may execute at any time, subject to precedence constraints, and timing constraints on predecessors and successors. Note that d_1 reads s_1, and d_2 reads s_2. To minimize jitter, both sensors are read between times 0 and 2 (for an explanation of jitter minimization in Giotto, see [HHK03]); thus the jobs $\mathit{true}(d_1)[0,7]$ and $\mathit{true}(d_2)[0,7]$ must start after time 2. Similarly, since sensor s_1 is read between times 6 and 7, $\mathit{true}(d_1)[1,7]$ must start after time 7. Finally, note that d_3 reads the output ports of t_1 and t_2

[1] What exactly the unit of time is, whether seconds, milliseconds, microseconds, etc., is not of importance in this report.

```
sensor
  port s_1 type int time 1
  port s_2 type int time 1
actuator
  port a type int init 0
input
  port i_1 type int
  port i_2 type int
output
  port o_1 type int init 0
  port o_2 type int init 0

task t_1 input i_1 output o_1 function f_1 time 1
task t_2 input i_2 output o_2 function f_2 time 1

driver d_1 source s_1 guard true destination i_1 function h_1 time 1
driver d_2 source s_2 guard true destination i_2 function h_2 time 2
driver d_3 source o_1, o_2 guard true destination a function h_3 time 1

mode m period 12 ports o_1, o_2
  frequency 2 invoke t_1 driver d_1
  frequency 1 invoke t_2 driver d_2
  frequency 2 update d_3

start m
```

Figure 1: Preemptible drivers program.

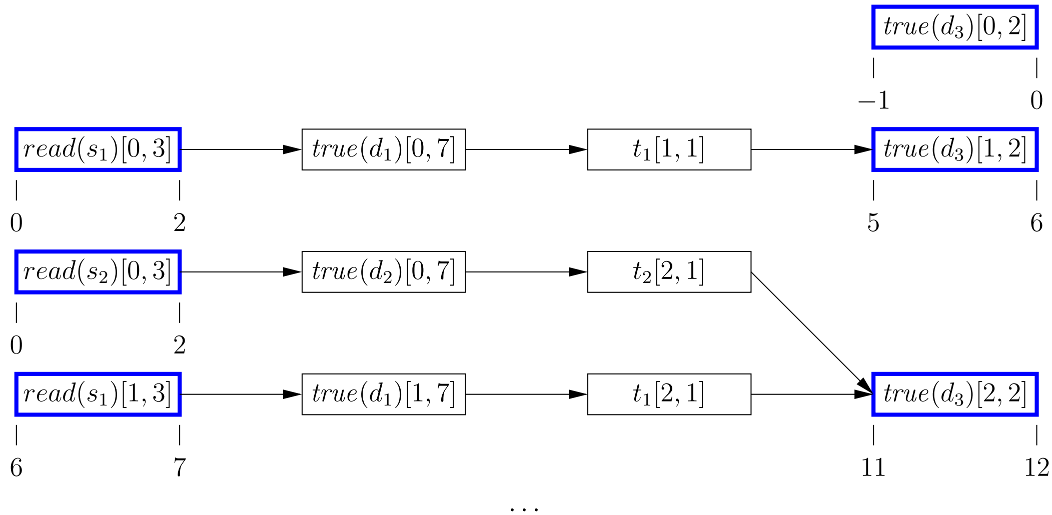

Figure 2: Preemptible drivers timing constraints.

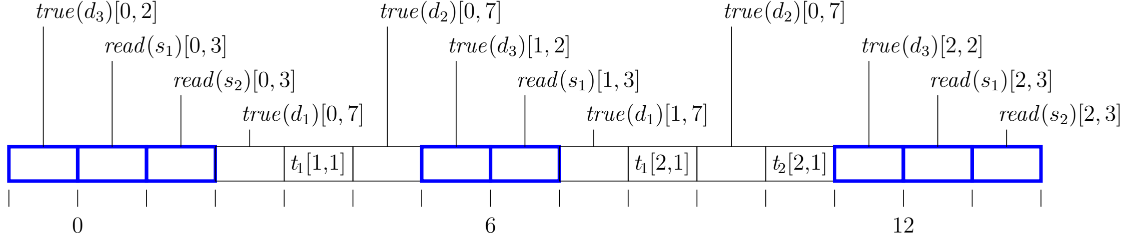

Figure 3: Preemptible drivers schedule.

between times 5 and 6, and between times 11 and 12. Thus, the job $t_1[1,1]$ must complete before time 5. Similarly, the jobs $t_2[2,1]$ and $t_1[2,1]$ must start after time 11. It may be verified that no schedule can meet these constraints unless the job $\mathit{true}(d_2)[i,7]$ is preempted, for $i = 0, 2, 4, \cdots$.

Figure 3 shows a schedule for the program of Figure 1. Note that d_2 finishes *half* of its execution between times 4 and 5. At time 5, d_2 is preempted by d_3, s_1, d_1, and t_1. Finally, at time 9, d_2 is able to finish. The schedule from time 11 to time 23 repeats the schedule from -1 to 11, with the indices of jobs incremented by 2. Similarly, the schedule from 23 to 34 repeats the schedule from -1 to 11, and so on forever.

2.2 Spillover

Figure 4 shows another Giotto program. Mode m invokes tasks t_1 and t_2 with frequency 2. Using driver d_1, t_1 reads sensor s and the output port o_2 of t_2. Task t_2 reads only s. Using drivers d_3 and d_4, respectively, mode m also updates actuators a_1 and a_2 with frequencies 1 and 2. Both d_3 and d_4 read the output port o_1 of task t_1. To read o_1, a_1 uses d_3, and a_2 uses d_4. There are two important timing requirements to note:

1. Because $\mathit{true}(d_1)[0,7]$ receives an input from $\mathit{read}(s)[0,3]$, it must begin before time 1. Since the output of $t_1[1,1]$ is read by actuator driver $\mathit{true}(d_4)[1,2]$, $t_1[1,1]$ must finish by time 10. In general, $\mathit{true}(d_1)[i,7]$ and $t_1[i+1,1]$ must begin after time $11i+1$ for $i = 0, 1, 2, \cdots$. Also, $\mathit{true}(d_1)[i,7]$ and $t_1[i+1,1]$ must finish by $11i+10$ for $i = 0, 2, 4, \cdots$, and by $11i+6$ for $i = 1, 3, 5, \cdots$.

2. Similarly, $\mathit{true}(d_2)[0,7]$ must begin after time 1. In general, $\mathit{true}(d_2)[i,7]$ and $t_2[i+1,1]$ must begin after time $11i+1$, for $i = 0, 1, 2, \cdots$. Note that the actuator drivers d_3 and d_4 read only t_1's output, not t_2's output. Since t_1 reads t_2's output, t_2 inherits its deadline from t_1. Thus, $\mathit{true}(d_2)[i,7]$ and $t_2[i+1,1]$ must finish by time $11i+17$ for $i = 0, 2, 4, \cdots$, and by $11i+21$ for $i = 1, 3, 5, \cdots$.

Under the assumption that sensors and actuators execute during times $[11i-5, 11i+1]$ for $i = 0, 2, 4, \cdots$, and during times $[11i-1, 11i+1]$ for $i = 1, 3, 5, \cdots$, it may be verified that no schedule can meet these constraints unless the job $t_2[i,1]$ finishes after its logical deadline at time $22i$ for $i = 2, 4, 6, \cdots$. We call this phenomenon *spillover*.

Figure 5 shows a schedule for the program of Figure 4. Note that the second invocation of t_2, $t_2[2,1]$, cannot complete before its logical deadline at time 22, because actuator drivers d_3 and d_4 need to execute. These actuator drivers need the output of $t_1[2,1]$, which is complete, so the fact that $t_2[2,1]$ is not complete does not cause a problem. Job $t_2[2,1]$ is able to complete at time 25, though its execution spills over into what is logically the second round of the Giotto program. The schedule from time 33 until time 55, most of which is not shown, repeats the schedule from 11 until 33, with the indices of jobs incremented by 2. Similarly, the schedule from time 55 until 77 repeats that from 11 until 33, and so on forever.

```
sensor
  port s type int time 1
actuator
  port a1 type int init 0
  port a2 type int init 0
input
  port i1 type int
  port i2 type int
output
  port o1 type int init 0
  port o2 type int init 0

task t1 input i1 output o1 function f1 time 1
task t2 input i2 output o2 function f2 time 4

driver d1 source s,o2 guard true destination i1 function h1 time 1
driver d2 source s guard true destination i2 function h2 time 1
driver d3 source o1 guard true destination a1 function h3 time 1
driver d4 source o1 guard true destination a2 function h4 time 4

mode m period 22 ports o1,o2
    frequency 2 invoke t1 driver d1
    frequency 2 invoke t2 driver d2
    frequency 1 update d3
    frequency 2 update d4

start m
```

Figure 4: Spillover program.

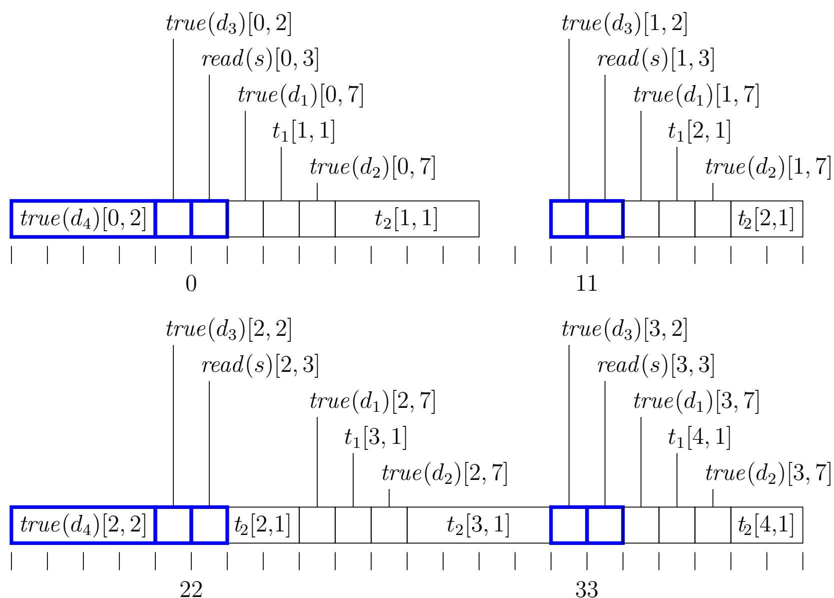

Figure 5: Spillover schedule.

3 Scheduling models

3.1 The three-field notation $\alpha \mid \beta \mid \gamma$

In scheduling theory, a standard notation is used to describe scheduling problems [LLK82, HLv97]. This notation has been in use since the early 1980s. By that time, the number of scheduling problems had grown so large (one estimate put the number at 4,536 [LLLK82]) that mathematical methods were needed to understand the relationships between the problems. The standard notation helps to classify the computational complexity of the problems, by making it apparent when one problem is more expressive than another. The standard notation consists of three fields, α, β, and γ, and is typically written $\alpha \mid \beta \mid \gamma$. The meaning of these fields is as follows:

- α describes the machines which are to be scheduled. For example, $\alpha = 1$ means a single-machine, $\alpha = P$ means parallel identical machines, and $\alpha = J$ means a job shop.[2]
- β describes task parameters and capabilities. To take two examples, $\beta = pmtn$ means that preemption is allowed, and $\beta = r_j; pmtn$ means that activities have release times and preemption is allowed.
- γ represents the cost function. For instance, $\gamma = C_{max}$ means the cost of a schedule is the maximum activity completion time (also called the *makespan*), and $\gamma = L_{max}$ means the cost is maximum activity lateness (which assumes a deadline has been given for each activity).

We now discuss how to choose appropriate values of α, β, and γ for the problem of scheduling single-mode Giotto programs on a single processor.

- Since this report is concerned with single-processor scheduling, $\alpha = 1$.
- The selection of β is more involved. An activity that is required to execute at a fixed time can be modeled with an appropriate choice of release time and deadline. For example, an actuator job j that must execute just before time 0, and that requires 1 unit of time, may be modeled by setting the release time of j equal to -1, and the deadline of j equal to 0. Thus, it is useful to have β include r_j and d_j (release times and deadlines). In addition, in order to model dataflow dependencies, β should include *prec* (precedences). Finally, β should include *pmtn*, not only because preemption is a common feature of real-time operating systems, but also because many problems that are otherwise computationally easy become hard when preemption is disallowed.[3]
- The choice of γ is simpler. Were we to adhere strictly to the standard notation, we would use the cost function $\gamma = L_{max}$ (or maximum lateness). However, we are really only concerned with determining whether there is a schedule for which $L_{max} \leq 0$, and synthesizing such a schedule if so. For this reason, we use the variant of the standard notation in which $\gamma = -$ [B$^+$01]. This variant asks whether there is a schedule in which every task finishes before its deadline.

In conclusion, the scheduling problem for single-mode, single-processor Giotto programs is similar to $1 \mid r_j; d_j; prec; pmtn \mid -$, which asks whether a set of activities with release times, deadlines, and precedence constraints is schedulable with preemption on a single machine such that all deadlines are met.

3.2 The problem $1 \mid r_j; d_j; prec; pmtn \mid -$

This similarity of single-mode Giotto scheduling for a single processor to $1 \mid r_j; d_j; prec; pmtn \mid -$ allows us to use an optimal algorithm for $1 \mid r_j; d_j; prec; pmtn \mid -$ as a starting point. This section precisely defines the problem $1 \mid r_j; d_j; prec; pmtn \mid -$, and presents an algorithm for it due to J. Błażewicz [Bła76].

[2] In parallel models, including $\alpha = P$, activities are allowed to migrate between machines. In shop models, including $\alpha = J$, a activity consists of a set of operations, each such operation being fixed to a particular machine. What we call *activities* are commonly called *jobs* or *tasks* in the scheduling literature. The latter two terms are already reserved for entities relating to a Giotto program (*tasks* and *jobs* are, respectively, programming-language-level and implementation-level entities that perform computation [HHK03]), so we use the term *activities* instead.

[3] For example, the problem $1 \mid r_j; pmtn \mid L_{max}$ is in P, whereas $1 \mid r_j \mid L_{max}$ is NP-hard [Len77].

Definition 1 ($1 \mid r_j; d_j; prec; pmtn \mid -$). An instance of $1 \mid r_j; d_j; prec; pmtn \mid -$ is a tuple $(A, t, r, d, \ll)$, where:

- A is a finite set, called the set of *activities*.
- $t : A \to \mathbb{Z}^{>0}$ is a function that assigns each activity a positive integer, called the activity's *computation time*.
- $r : A \to \mathbb{Z}^{\geq 0}$ is a function that assigns each activity a nonnegative integer, called the activity's *release time*.
- $d : A \to \mathbb{Z}^{>0}$ is a function that assigns each activity a positive integer, called the activity's *deadline*.[4]
- $\ll \subseteq A \times A$ is a relation on A, called the *precedence relation*. We shall normally write $a_1 \ll a_2$ instead of $(a_1, a_2) \in \ll$. □

Several remarks are in order. For Giotto, an activity might be an invocation of a sensor, an actuator driver, a task driver, or a task. However, as far as $1 \mid r_j; d_j; prec; pmtn \mid -$ is concerned, an activity is simply something that takes time. A precedence constraint $a_1 \ll a_2$ requires that a_1 finish before a_2 can begin. If $a_1 \ll a_2$, we say that a_1 is a *predecessor* of a_2. An activity a may execute at any time after $r(a)$, as long as all its predecessors are complete, and a must finish before $d(a)$. It will follow from the fact that $t(a) > 0$ for each $a \in A$, and from Definition 2, that an instance of $1 \mid r_j; d_j; prec; pmtn \mid -$ is feasible only if $\ll$ is acyclic.

Definition 2 (schedule, feasibility). A *schedule* S is a pair (I, e), where:

- I is a finite set of intervals of the real line $\mathbb{R}$. Each interval in I must be nonempty and of the form (ℓ, r), i.e., left-open and right-open. Intervals in I must also be non-overlapping; i.e., if $i_1, i_2 \in I$ and $i_1 \neq i_2$, then $i_1 \cap i_2 = \emptyset$.
- $e : I \to A$ is a function that assigns an activity $e(i)$ to each interval i. We say that the activity $e(i)$ is *executed* in interval i.

Given an activity a in the range of e, let $I[a]$ be the set of intervals in which a is executed, i.e., the set $\{i \in I \mid e(i) = a\}$. Given a schedule S and an activity a, we define several functions:

- The *start time* of activity a in S, $\mathcal{S}_S(a)$, is $\inf_{(\ell,r)\in I[a]} \ell$.
- The *finish time* of a in S, $\mathcal{F}_S(a)$, is $\sup_{(\ell,r)\in I[a]} r$.
- The *total execution time* of a in S, $\mathcal{T}_S(a)$, is $\sum_{(\ell,r)\in I[a]} r - \ell$.

We say that schedule S *satisfies* (or is *feasible* for) problem instance $P = (A, t, r, d, \ll)$ if the following conditions are met:

- For each activity $a \in A$, $r(a) \leq \mathcal{S}_S(a)$, $\mathcal{F}_S(a) \leq d(a)$, and $t(a) = \mathcal{T}_S(a)$.
- For each $a_1, a_2 \in A$ such that $a_1 \ll a_2$, $\mathcal{F}_S(a_1) \leq \mathcal{S}_S(a_2)$.

We say that P is *feasible* if there is a schedule S that satisfies P. □

In 1976, J. Błażewicz developed a polynomial-time algorithm EDF$^{\prec}$ that, given an instance P of $1 \mid r_j; d_j; prec; pmtn \mid -$, finds a schedule satisfying the constraints of P if one exists. Let $\ll^*$ (respectively, $\ll^+$) denote the transitive reflexive (respectively, transitive) closure of $\ll$. EDF$^{\prec}$ relies on transitive release time and deadline functions r^* and d^*, defined by:

$$r^*(a) = \max_{\{a' \mid a' \ll^* a\}} r(a')$$
$$d^*(a) = \min_{\{a' \mid a \ll^* a'\}} d(a')$$

[4]For simplicity, we require that $t(a)$, $r(a)$, and $d(a)$ are integers, for each activity $a \in A$. The results of this chapter would continue to hold if these quantities were allowed to be rational.

We say that activity a is *enabled* at time τ if $r^*(a) \leq \tau$ and, for all a' such that $a' \ll^+ a$, a' has executed for at least $t(a')$ time units up to τ. EDF$^\prec$ schedules activities according to the following rule:

$$\text{At each time } \tau\text{, execute an enabled activity } a \text{ with minimal } d^*(a) \text{ value.} \tag{1}$$

The $O(|A|^2)$ running time of EDF$^\prec$ was reduced to $O(|A| \log |A|)$ by [Kim94]. A clear proof of the optimality of EDF$^\prec$ may be found in [Bru01].[5]

3.3 A periodic version of $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$

The problem $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$ is a not a perfect match for single-mode, single-processor Giotto programs, since such programs have infinite, periodic streams of activities. In this section, we define a periodic version of $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$. We then develop a scheduling algorithm for this periodic version.

Definition 3 ($1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$). An instance of $1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$ is a tuple $P = (A, t, r, d, \ll, \Pi)$, where:

- A is the union of disjoint sets $A_0, A_1, \cdots$, each with the same number $n \in \mathbb{Z}^{>0}$ of elements. For notational convenience, let $a\langle i, 1\rangle, \cdots, a\langle i, n\rangle$ be the members of A_i.
- $\Pi \in \mathbb{Z}^{>0}$ is called the *period*.
- The functions t, r, d, and the relation $\ll$ are defined as they were in Definition 1, and must satisfy the following additional conditions:
  - $t(a\langle i, k\rangle) = t(a\langle 0, k\rangle)$ for all $i \in [1 .. \infty]$ and $k \in [1 .. n]$.
  - $r(a\langle i, k\rangle) = r(a\langle 0, k\rangle) + i\Pi$, and $d(a\langle i, k\rangle) = d(a\langle 0, k\rangle) + i\Pi$, for all $i \in [1 .. \infty]$ and $k \in [1 .. n]$.
  - For all $k \in [1 .. n]$,
    $$r(a\langle 0, k\rangle) \in [0 .. \Pi - 1] \tag{2}$$
    (This requirement, though not essential, simplifies the proofs below.)
  - The precedence relation $\ll$ satisfies the uniformity condition:
    $$a\langle i_1, k_1\rangle \ll a\langle i_2, k_2\rangle \quad \text{iff} \quad a\langle 0, k_1\rangle \ll a\langle i_2 - i_1, k_2\rangle \tag{3}$$
    for all $i_1, i_2 \in [0 .. \infty]$ and all $k_1, k_2 \in [1 .. n]$. (Equivalently, $a\langle i_1, k_1\rangle \ll a\langle i_2, k_2\rangle$ iff $a\langle i_1 - i_1, k_1\rangle \ll a\langle i_2 - i_1, k_2\rangle$.)
  - If $a\langle i_1, k_1\rangle \ll a\langle i_2, k_2\rangle$ then $i_1 \leq i_2$ for all $i_1, i_2 \in [0 .. \infty]$ and all $k_1, k_2 \in [1 .. n]$. □

Definition 4 (schedule, feasibility). This definition follows Definition 2, with a modification to account for the infinite nature of the problem instance P. A *schedule* S is a pair (I, e), where I is defined as it was in Definition 2, except that I need not be finite; and e is defined as it was in Definition 2. Further, $\mathcal{S}_S$, $\mathcal{F}_S$, $\mathcal{T}_S$, *satisfaction*, and *feasibility* are defined as they were in Definition 2.[6] □

The rest of this section extends EDF$^\prec$ to our new setting. It is not immediately obvious how to do so, since EDF$^\prec$ works on finite problem instances. Section 3.3.1 develops a necessary condition on feasibility for our periodic problem. Section 3.3.2 extends this condition into a necessary and sufficient condition that provides us with an optimal, polynomial-time algorithm for $1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$.

---

[5] A variant of EDF$^\prec$ is presented in [SBS95], in which $r^*(a)$ and $d^*(a)$ are defined by:

$$\begin{aligned} r^*(a) &= \max(r(a), \max_{\{a' \mid a' \ll^+ a\}} r^*(a') + t(a)) \\ d^*(a) &= \min(d(a), \min_{\{a' \mid a \ll^+ a'\}} d^*(a') - t(a)) \end{aligned}$$

and activities are scheduled according to the rule (1). This variant is also optimal.

[6] Note that $\mathcal{S}_S(a)$ may be $-\infty$, $\mathcal{F}_S(a)$ may be $+\infty$, and $\mathcal{T}_S(a)$ may be ∞. A schedule in which any of these quantities is infinite is of no interest, since it cannot satisfy P.

3.3.1 A necessary condition on feasibility for $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$

We begin our analysis with a definition of *active* schedules, in which the processor eagerly executes any available activity.

Definition 5 (active schedule). Let $S = (I, e)$ be a schedule. We say that $\tau \in \mathbb{R}$ is an *idle time* if $\tau \notin [\ell, r]$ for any $(\ell, r) \in I$. An activity $a \in A$ is *complete* at τ if

$$t(a) \leq \sum_{i \in I[a]} |i \cap [-\infty, \tau]|$$

S is *active* if (1) for all idle times τ, there is no activity $a \in A$ such that $r^*(a) \leq \tau$ and a is not complete at τ, and (2) for every activity $a \in A$, $\mathcal{T}_S(a) = t(a)$. □

Intuitively, a feasible schedule is active if no activity can be executed earlier without some other activity being executed later. In the remainder of this section, we consider only active schedules. The justification for this restriction is provided by the following proposition, which is evident:

Proposition 6. *If P is a feasible instance of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$, then there exists an active schedule that satisfies P.*

Active schedules have the following convenient property: for any two active schedules, the amount of computation pending at time τ (the amount of computation released but not completed) is the same.

We now develop a condition that any instance P of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$ must satisfy in order to be feasible. This condition centers around the notion of a *rest point*, an instant when no computation is pending. We will establish several lemmas concerning rest points, leading up to Theorem 13: P is feasible only if the set $[\Pi \mathrel{..} 2\Pi]$ contains a rest point.

Definition 7 (pending computation function, rest point). Let

$$T(i) = \sum_{\{a \in A \;\mid\; r^*(a) \,=\, i\}} t(a)$$

$T(i)$ is the amount of computation whose transitive release time is i. Note that $T(i) = T(i{+}\Pi)$ for $i \in [0 \mathrel{..} \infty]$. We define the *pending computation function* $p : \mathbb{Z}^{\geq 0} \rightarrow \mathbb{Z}^{\geq 0}$ as follows:

- $p(0) = T(0)$.
- For $i > 0$,

$$p(i) = T(i) + \begin{cases} p(i-1) - 1 & \text{if } p(i-1) > 0 \\ 0 & \text{if } p(i-1) = 0 \end{cases}$$

Let $p^- : \mathbb{Z}^{\geq 0} \rightarrow \mathbb{Z}^{\geq 0}$ be defined by:

- $p^-(0) = 0$.
- For $i > 0$,

$$p^-(i) = \begin{cases} p(i-1) - 1 & \text{if } p(i-1) > 0 \\ 0 & \text{if } p(i-1) = 0 \end{cases}$$

Note that $p^-(i) = p(i) - T(i)$. We say that $i \in \mathbb{Z}^{\geq 0}$ is a *rest point* if $p^-(i) = 0$.[7] Note that 0 is a rest point, since $p^-(0) = p(0) - T(0) = T(0) - T(0)$. □

Example 8. We use an example to illustrate the concept of a rest point. Suppose that

$$T(0) = 5 \qquad T(5) = 3 \qquad T(6) = 1 \qquad T(9) = 3 \qquad T(12) = 2 \qquad T(14) = 1$$

and that $T(i) = 0$ for $i \in [0 \mathrel{..} 14] \setminus \{0, 5, 6, 9, 12, 14\}$. Figure 6 presents an graph of the pending computation $p(i)$. Times $i \in \{0, 5, 9, 12, 14\}$ are rest points. All other $i \in [0 \mathrel{..} 14]$ are not rest points. □

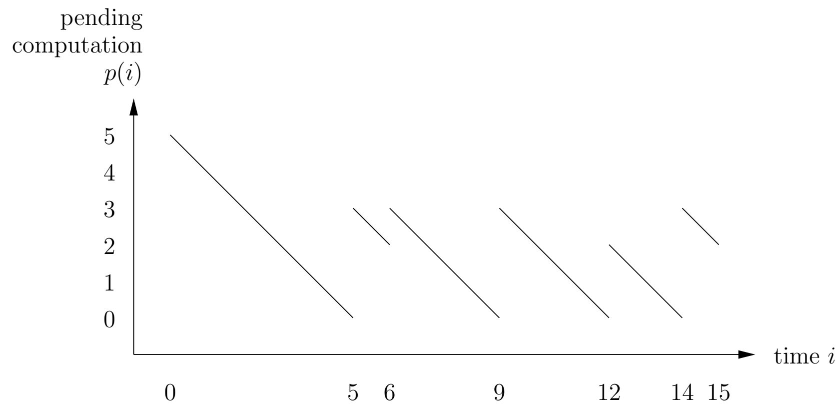

Figure 6: An illustration of the concept of rest points.

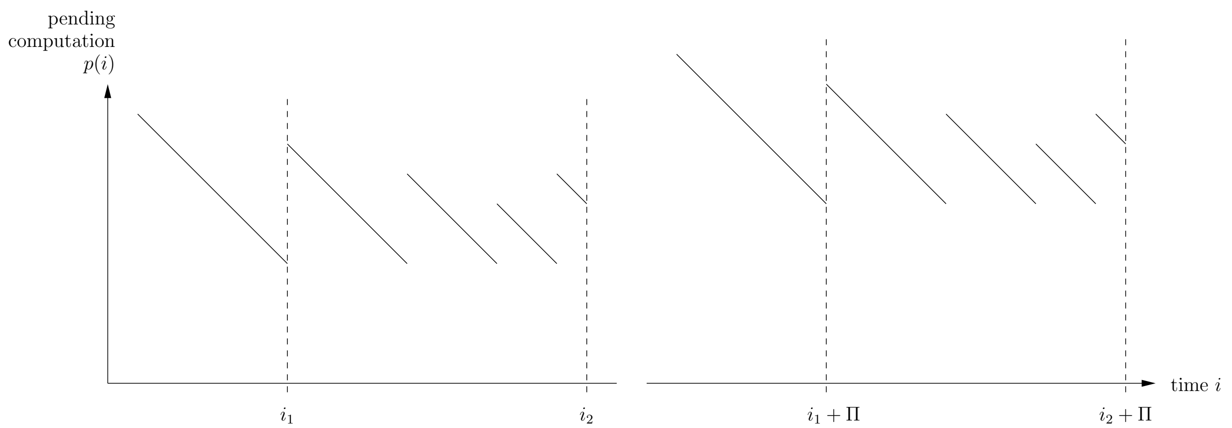

Figure 7: A diagram for Lemma 9.

We next derive a simple but useful fact about the pending computation function.

Lemma 9. Let $i_1 < i_2$ be nonnegative integers. If $p(i) > 0$ for $i \in [i_1 \,..\, i_2] \cup [i_1 + \Pi \,..\, i_2 + \Pi]$, then for $i \in [i_1 \,..\, i_2]$,

$$p(i + \Pi) = p(i) - p^-(i_1) + p^-(i_1 + \Pi) \tag{4}$$
$$p(i + \Pi) = p(i) - p(i_1) + p(i_1 + \Pi) \tag{5}$$

Figure 7 may aid the reader in understanding the significance of the Lemma.

Proof. Since $p(i) > 0$ for $i \in [i_1 \,..\, i_2]$,

$$p(i) = p^-(i_1) - (i - i_1) + \sum_{k=i_1}^{i} T(k) \tag{6}$$

Similarly, since $p(i + \Pi) > 0$ for $i \in [i_1 \,..\, i_2]$,

$$p(i + \Pi) = p^-(i_1 + \Pi) - ((i + \Pi) - (i_1 + \Pi)) + \sum_{k=i_1+\Pi}^{i+\Pi} T(k) \tag{7}$$

Since $\sum_{k=i_1}^{i} T(k) = \sum_{k=i_1+\Pi}^{i+\Pi} T(k)$, (7) simplifies to

$$p(i + \Pi) = p^-(i_1 + \Pi) - (i - i_1) + \sum_{k=i_1}^{i} T(k) \tag{8}$$

Comparing (6) and (8), we see that

$$p(i + \Pi) = p(i) - p^-(i_1) + p^-(i_1 + \Pi) \tag{9}$$

which yields (4). Since $p(i) = p^-(i) + T(i)$, and $T(i) = T(i + \Pi)$,

$$\begin{aligned} -p^-(i_1) + p^-(i_1 + \Pi) &= T(i_1) - p(i_1) - T(i_1 + \Pi) + p(i_1 + \Pi) \\ &= -p(i_1) + p(i_1 + \Pi) \end{aligned} \tag{10}$$

From (9) and (10),

$$p(i + \Pi) = p(i) - p(i_1) + p(i_1 + \Pi)$$

which yields (5). □

Lemma 10. If there is no rest point in $[\Pi \,..\, 2\Pi]$, then $p(2\Pi) > p(\Pi)$.

Proof. Let r be the latest rest point in $[0 \,..\, \Pi]$. Since 0 is a rest point, there is at least one such rest point. We now verify that the conditions of Lemma 9 are satisfied for $i_1 = r$ and $i_2 = \Pi$. Since r is the *latest* rest point, $p(i) > 0$ for $i \in [r \,..\, \Pi]$. Since there is no rest point in $[\Pi \,..\, 2\Pi]$, *a fortiori* $p(i) > 0$ for $i \in [\Pi + r \,..\, 2\Pi]$. The conditions of Lemma 9 are thus fulfilled. Applying (4) with $i = \Pi$, we obtain:

$$p(2\Pi) = p(\Pi) - p^-(r) + p^-(\Pi + r) \tag{11}$$

Since r is a rest point, $p^-(r) = 0$, and since $\Pi + r$ is not a rest point, $p^-(\Pi + r) > 0$. Thus, by equation (11), $p(2\Pi) > p(\Pi)$. □

Lemma 11. Let k be a member of $\mathbb{Z}^{\geq 0}$. If $p(i) > 0$ for $i \in [k\Pi \,..\, (k + 2)\Pi]$, then

$$p((k + 2)\Pi) = 2p((k + 1)\Pi) - p(k\Pi)$$

[7]If p were defined on $\mathbb{R}^{\geq 0}$ instead of $\mathbb{Z}^{\geq 0}$, then a rest point would be an instant τ at which $\lim_{\tau' \to \tau^-} p(\tau') = 0$. The definition of p as a function on $\mathbb{Z}^{\geq 0}$ is simpler, though it makes the definition of a rest point less intuitive.

Proof. Let $i_1 = k\Pi$ and $i_2 = (k+1)\Pi$. The conditions of Lemma 9 are fulfilled. Setting $i = (k+1)\Pi$, and applying (5), we obtain that

$$p((k+1)\Pi + \Pi) = p((k+1)\Pi) - p(k\Pi) + p(k\Pi + \Pi)$$

which yields our result. □

Lemma 12. If there is no rest point in $[\Pi \mathrel{..} 2\Pi]$, then for $k = 1, 2, \cdots$:

- there is no rest point in $[k\Pi \mathrel{..} (k+1)\Pi]$, and
- $p((k+1)\Pi) = p(\Pi) + k(p(2\Pi) - p(\Pi))$.

Proof. For $k = 1$, the lemma reduces to the claims (1) that there is no rest point in $[\Pi \mathrel{..} 2\Pi]$, which is true by assumption, and (2) that $p(2\Pi) = p(\Pi) + (p(2\Pi) - p(\Pi))$, which is trivially true. For the induction step, suppose as induction hypothesis that the lemma is true for all $j \le k$. We need to show that the lemma remains true for $k+1$.

By Lemma 10, $p(2\Pi) > p(\Pi)$. From this, and the induction hypothesis for k, it follows that $p((k+1)\Pi) > p(k\Pi)$. Also, by the induction hypothesis for k, there is no rest point in $[k\Pi \mathrel{..} (k+1)\Pi]$. Thus, there is no rest point in $[(k+1)\Pi \mathrel{..} (k+2)\Pi]$, since in this range the initial pending computation is greater. Since $[k\Pi \mathrel{..} (k+2)\Pi]$ contains no rest point, $p(\ell) > 0$ for $\ell \in [k\Pi \mathrel{..} (k+2)\Pi]$. By Lemma 11,

$$p((k+2)\Pi) = 2p((k+1)\Pi) - p(k\Pi) \tag{12}$$

By the induction hypothesis,

$$p((k+1)\Pi) = p(\Pi) + k(p(2\Pi) - p(\Pi)) \tag{13}$$

Also by the induction hypothesis,

$$p((k+1)\Pi) - p(k\Pi) = p(2\Pi) - p(\Pi) \tag{14}$$

Thus,

$$\begin{aligned} p((k+2)\Pi) &= \Big[p((k+1)\Pi)\Big] \quad + \quad \Big[p((k+1)\Pi) - p(k\Pi)\Big] && \text{(by (12))} \\ &= \Big[p(\Pi) + k(p(2\Pi) - p(\Pi))\Big] \quad + \quad \Big[p(2\Pi) - p(\Pi)\Big] && \text{(by (13) and (14))} \\ &= p(\Pi) + (k+1)(p(2\Pi) - p(\Pi)) \end{aligned}$$

as desired. □

If there is no rest point in $[\Pi \mathrel{..} 2\Pi]$, then Lemma 12 states that at the successive times $2\Pi, 3\Pi, \cdots$, the amount of pending computation increases by $p(2\Pi) - p(\Pi)$. By Lemma 10, the quantity $p(2\Pi) - p(\Pi)$ is positive. Thus, the amount of pending computation at times $i\Pi$ increases without bound. Intuitively, this indicates that eventually some activity must be late. The following theorem confirms this intuition.

Theorem 13. An instance $P = (A, t, r, d, \ll)$ of $1 \mid r_j;\ d_j;\ \mathit{prec};\ \mathit{pmtn};\ \mathit{period} \mid -$ is feasible only if the set $[\Pi \mathrel{..} 2\Pi]$ contains a rest point.

Proof. We will use Lemma 12 to show that if $[\Pi \mathrel{..} 2\Pi]$ contains no rest point, then P is infeasible. Let $D = \max_{a \in A} (d^*(a) - r^*(a))$. If, at time $i\Pi$, some activity a with $r^*(a) \le i\Pi - D$ is not complete, then a or some successor of a has missed its deadline. We now examine how large i has to be so that the following stronger condition attains:

$$\text{At time } i\Pi\text{, some activity } a \text{ with} \quad r^*(a) \le \left(i - \left\lceil \frac{D}{\Pi} \right\rceil\right)\Pi \quad \text{is not complete.}$$

Let $T = \sum_{a \in A_0} t(a)$. By the pigeonhole principle, if $p(i\Pi) > T$, then some activity a with $r^*(a) < i\Pi$ is not complete at time $i\Pi$. Similarly, if $p(i\Pi) > kT$, then some activity a with $r^*(a) < (i-k+1)\Pi$ is not complete at time $(i-k+1)\Pi$. Thus, we need to choose k such that

$$i - k + 1 \leq \left(i - \left\lceil \frac{D}{\Pi} \right\rceil\right)$$

It suffices to set

$$k \geq \left\lceil \frac{D}{\Pi} \right\rceil + 1$$

We now choose i such that $p(i\Pi) > kT$. By Lemma 12, $p(i\Pi) = p(\Pi) + (i-1)(p(2\Pi) - p(\Pi))$. Setting $p(\Pi) + (i-1)(p(2\Pi) - p(\Pi)) > kT$, and solving for i, we obtain

$$i > \frac{kT + p(2\Pi) - 2p(\Pi)}{p(2\Pi) - p(\Pi)}$$

Choosing any i satisfying this inequality (e.g., one plus the ceiling of the right hand side) will suffice. We have shown that by time $i\Pi$ some activity will have missed its deadline. Thus, P is infeasible. □

3.3.2 A necessary and sufficient condition on feasibility for $1 \mid r_j; d_j; prec; pmtn; period \mid -$

The previous section presented a necessary condition on feasibility for an instance P of $1 \mid r_j; d_j; prec; pmtn; period \mid -$, namely, that there be a rest point in $[\Pi \,..\, 2\Pi]$. However, P may have such a rest point, but still be infeasible; this occurs if the activities cannot be scheduled to meet their deadlines. In this section, we extend Theorem 13 into a necessary and sufficient condition on feasibility (Theorem 16). This extension relies on an attractive property of the pending computation function, namely, that if $i \in [\Pi \,..\, 2\Pi]$ is a rest point, then p "looks the same" on the intervals

$$[i - \Pi \,..\, i-1], \quad [i \,..\, i+\Pi-1], \quad [i+\Pi \,..\, i+2\Pi-1], \quad \cdots$$

In other words, p is periodic, with period Π, beginning at $i - \Pi$, as we now show.

Lemma 14. If there is a rest point in $i \in [\Pi \,..\, 2\Pi]$, then $p(k) = p(k+\Pi)$ for $k \in [i-\Pi \,..\, \infty]$.

Proof. Recall that $T(\cdot)$ is periodic, i.e., $T(k) = T(k-\Pi)$ for $k \in [\Pi \,..\, \infty]$. Note that $p(0) \leq p(\Pi)$, since

$$p(0) = T(0) = T(\Pi) \leq T(\Pi) + p^-(\Pi) = p(\Pi)$$

Suppose that $i \in [\Pi \,..\, 2\Pi]$ is a rest point. Then $p^-(i-\Pi) = 0$, since $T(\cdot)$ is periodic and the initial pending computation $p(0)$ is greater than $p(\Pi)$. Since $T(\cdot)$ is periodic, and both $i - \Pi$ and i are rest points,

$$p(k) = p(k+\Pi) \text{ for } k \in [i-\Pi \,..\, i-1]$$

A simple argument by induction establishes that for all $\ell = 1, 2, \cdots$,

$$p(k) = p(k+\ell\Pi) \text{ for } k \in [i-\Pi \,..\, i-1]$$ □

A similar argument establishes the following lemma.

Lemma 15. Let k be a positive integer, and let i be a member of $[0 \,..\, \Pi - 1]$. If $i + k\Pi$ is a rest point, then $i + k'\Pi$ is a rest point for *all nonnegative* integers $k' = 0, 1, \cdots$.

Intuitively, Lemmas 14 and 15 allow the timeline to be divided into sections $[i-\Pi \,..\, i-1]$, $[i \,..\, i+\Pi-1]$, etc., each of length Π; these sections may be scheduled using EDF$^{\prec}$. Indeed, suppose that (1) there is a rest point i in $[\Pi \,..\, 2\Pi]$, and (2) EDF$^{\prec}$ produces a feasible schedule S for activities released in $[i-\Pi \,..\, i-1]$. Under these conditions, one can create a feasible schedule for all of P by "pasting together" successive copies of S, as the following theorem shows.

Theorem 16. An instance P of $1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$ is feasible if and only if

1. there is a rest point i in $[\Pi \mathinner{..} 2\Pi]$, and
2. $\mathrm{EDF}^{\prec}$ produces a feasible schedule for activities a with $r^*(a) \in [i - \Pi \mathinner{..} i - 1]$.

Proof. If the first condition does not hold, then Theorem 13 shows that P is infeasible. If the first condition holds but the second does not, then since $\mathrm{EDF}^{\prec}$ is optimal, P is infeasible. We have established the "only if" part. For the "if" part, suppose that $i \in [\Pi \mathinner{..} 2\Pi]$ is a rest point, and that $\mathrm{EDF}^{\prec}$ produces a feasible schedule $S = (I, e)$ for activities a with $r^*(a) \in [i - \Pi \mathinner{..} i - 1]$. For the "if" part, suppose that $i \in [\Pi \mathinner{..} 2\Pi]$ is a rest point, and that $\mathrm{EDF}^{\prec}$ produces a feasible schedule $S = (I, e)$ for activities a with $r^*(a) \in [i - \Pi \mathinner{..} i - 1]$. We use (I, e) to construct an infinite sequence (I_k, e_k) of schedules: for $k = 0, 1, \cdots$, let

- $I_k = \{(\ell + k\Pi, r + k\Pi) \mid (\ell, r) \in I\}$.
- $e_k(\ell + k\Pi, r + k\Pi) = a\langle m + k, n\rangle$, where $a\langle m, n\rangle = e(\ell, r)$.

Since (I, e) is feasible for activities a with $r^*(a) \in [i - \Pi \mathinner{..} i - 1]$, (I_k, e_k) is feasible for activities a with $r^*(a) \in [i - \Pi + k\Pi \mathinner{..} i - 1 + k\Pi]$.

In addition, we need to construct a schedule (I_{-1}, e_{-1}) for activities a with $r^*(a) \in [0 \mathinner{..} i - \Pi - 1]$. A technical point is that for these activities we must do something slightly different from the above, since I may contain intervals that intersect $[0, \Pi]$ (subtracting Π from these intervals, as suggested by the above definition of I_k, would amount to scheduling $[-\Pi, 0]$). Similarly, the activities executed in I may include members of A_0 (subtracting 1 from the first index of these activities would yield activities in the nonexistent set A_{-1}). Fortunately, all members of A_1 are executed after Π, since $a \in A_1$ implies $r^*(a) \geq \Pi$. We therefore let

- $I_{-1} = \{(\ell - \Pi, r - \Pi) \mid (\ell, r) \in I \text{ and } e(\ell, r) \in A_1\}$.
- $e_{-1}(\ell - \Pi, r - \Pi) = a\langle m - 1, n\rangle$, where $a\langle m, n\rangle = e(\ell, r) \in A_1$.

It is straightforward to verify that (I_{-1}, e_{-1}) is feasible for activities a with $r^*(a) \in [0 \mathinner{..} i - \Pi - 1]$. Finally, let $I_\infty = \bigcup_{k=-1}^{\infty} I_k$, let $e_\infty = \bigcup_{k=-1}^{\infty} e_k$, and let $S_\infty = (I_\infty, e_\infty)$. (I_∞, e_∞) satisfies P, thus establishing the "if" part. □

Example 17. We now present an example to illustrate how Theorem 16 schedules an instance P of $1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$. (This example is the problem instance generated by the Giotto program of Figure 4. Section 4 explains how to generate an instance of $1 \mid r_j;\ d_j;\ prec;\ pmtn;\ period \mid -$ from a Giotto program.) Let $P = (A, t, r, d, \ll, \Pi)$, where:

- The set of activities, A, is $\bigcup_{k=0}^{\infty} A_k$, where $A_k = \{a\langle k, \ell\rangle \mid \ell \in [1 \mathinner{..} 13]\}$.[8]
- For each activity a, the execution time of a, $t(a)$, is:

$$
\begin{array}{lll}
t(a\langle k,1\rangle) = 1 & t(a\langle k,2\rangle) = 4 & t(a\langle k,3\rangle) = 1 \\
t(a\langle k,4\rangle) = 1 & t(a\langle k,5\rangle) = 1 & t(a\langle k,6\rangle) = 1 \\
t(a\langle k,7\rangle) = 4 & t(a\langle k,8\rangle) = 1 & t(a\langle k,9\rangle) = 1 \\
t(a\langle k,10\rangle) = 1 & t(a\langle k,11\rangle) = 1 & t(a\langle k,12\rangle) = 1 \\
t(a\langle k,13\rangle) = 4 & &
\end{array}
$$

[8] The correspondence between these activities and the activities generated by the program of Figure 4 is as follows:

$$
\begin{array}{lll}
a\langle k,1\rangle = \mathit{true}(d_3)[2k,2] & a\langle k,2\rangle = \mathit{true}(d_4)[2k,2] & a\langle k,3\rangle = \mathit{read}(s)[2k,3] \\
a\langle k,4\rangle = \mathit{true}(d_1)[2k,7] & a\langle k,5\rangle = \mathit{true}(d_2)[2k,7] & a\langle k,6\rangle = t_1[2k+1,1] \\
a\langle k,7\rangle = t_2[2k+1,1] & a\langle k,8\rangle = \mathit{true}(d_3)[2k+1,2] & a\langle k,9\rangle = \mathit{read}(s)[2k+1,3] \\
a\langle k,10\rangle = \mathit{true}(d_1)[2k+1,7] & a\langle k,11\rangle = \mathit{true}(d_2)[2k+1,7] & a\langle k,12\rangle = t_1[2k+2,1] \\
a\langle k,13\rangle = t_2[2k+2,1] & &
\end{array}
$$

- For each activity a, the release time of a, $r(a)$, is:

$$\begin{array}{lll} r(a\langle k,1\rangle) = k\Pi & r(a\langle k,2\rangle) = k\Pi & r(a\langle k,3\rangle) = k\Pi+5 \\ r(a\langle k,4\rangle) = k\Pi+5 & r(a\langle k,5\rangle) = k\Pi+5 & r(a\langle k,6\rangle) = k\Pi+5 \\ r(a\langle k,7\rangle) = k\Pi+5 & r(a\langle k,8\rangle) = k\Pi+15 & r(a\langle k,9\rangle) = k\Pi+16 \\ r(a\langle k,10\rangle) = k\Pi+16 & r(a\langle k,11\rangle) = k\Pi+16 & r(a\langle k,12\rangle) = k\Pi+16 \\ r(a\langle k,13\rangle) = k\Pi+16 & & \end{array}$$

- For each activity a, the deadline of a, $d(a)$, is:

$$\begin{array}{lll} d(a\langle k,1\rangle) = k\Pi+5 & d(a\langle k,2\rangle) = k\Pi+5 & d(a\langle k,3\rangle) = k\Pi+6 \\ d(a\langle k,4\rangle) = k\Pi+16 & d(a\langle k,5\rangle) = k\Pi+27 & d(a\langle k,6\rangle) = k\Pi+16 \\ d(a\langle k,7\rangle) = k\Pi+27 & d(a\langle k,8\rangle) = k\Pi+16 & d(a\langle k,9\rangle) = k\Pi+17 \\ d(a\langle k,10\rangle) = k\Pi+27 & d(a\langle k,11\rangle) = k\Pi+38 & d(a\langle k,12\rangle) = k\Pi+27 \\ d(a\langle k,13\rangle) = k\Pi+38 & & \end{array}$$

- The following precedence constraints comprise $\ll$:

$$\begin{array}{lll} a\langle k,3\rangle \ll a\langle k,4\rangle & a\langle k,3\rangle \ll a\langle k,5\rangle & a\langle k,4\rangle \ll a\langle k,6\rangle \\ a\langle k,5\rangle \ll a\langle k,7\rangle & a\langle k,6\rangle \ll a\langle k,8\rangle & a\langle k,7\rangle \ll a\langle k,10\rangle \\ a\langle k,9\rangle \ll a\langle k,10\rangle & a\langle k,9\rangle \ll a\langle k,11\rangle & a\langle k,10\rangle \ll a\langle k,12\rangle \\ a\langle k,11\rangle \ll a\langle k,13\rangle & a\langle k,12\rangle \ll a\langle k+1,1\rangle & a\langle k,12\rangle \ll a\langle k+1,2\rangle \\ a\langle k,13\rangle \ll a\langle k+1,4\rangle & & \end{array}$$

  These precedence constraints are illustrated in Figure 8.

- Finally, $\Pi = 22$.

Figure 9 presents the pending computation function p for our example; p is defined by:

$$p(k) = \begin{cases} 5-k & \text{for } k \in [0 \mathrel{..} 4] \\ 8-(k-5) & \text{for } k \in [5 \mathrel{..} 13] \\ 0 & \text{for } k = 14 \\ 1 & \text{for } k = 15 \\ 8-(k-16) & \text{for } k \in [16 \mathrel{..} 21] \\ 7-(k-22) & \text{for } k \in [22 \mathrel{..} 26] \\ 10-(k-27) & \text{for } k \in [27 \mathrel{..} 36] \\ p(k-22) & \text{for } k \in [37 \mathrel{..} \infty] \end{cases}$$

The first rest point in $[\Pi \mathrel{..} 2\Pi]$ is at $i = 37$. Note that p is periodic, with period $\Pi = 22$, starting at $i-\Pi = 15$, as claimed by Lemma 14. Figure 10 presents a feasible schedule for activities a with $r^*(a) \in [i-\Pi \mathrel{..} i-1]$. Figure 11 shows a prefix of the schedule for P that is produced by the proof of Theorem 16. This schedule satisfies P, as desired. □

4 From Giotto programs to instances of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$

In this section, we will obtain a pseudopolynomial-time schedule synthesis algorithm for a class of single-mode Giotto programs. Our strategy will be to generate an instance of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$ given a program in this class, and then to apply the scheduling algorithm of Section 3.3.2 to this instance.

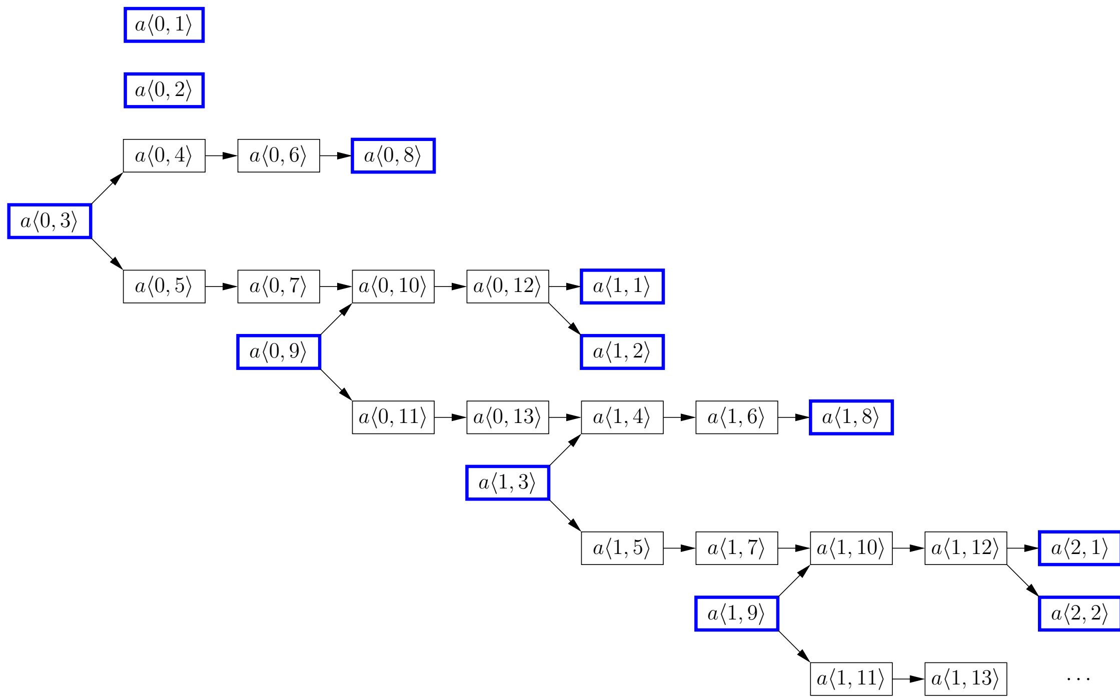

Figure 8: Precedence constraints for Example 17.

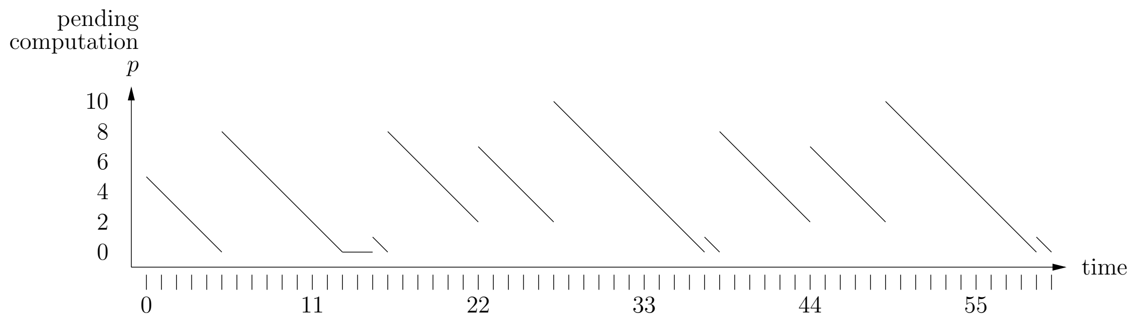

Figure 9: Pending computation function p for Example 17. Times 5, 13, 14, 15, 37, 38, 59, and 60 are rest points. For $i \in [0 .. \infty]$, times $15 + 22i$ and $16 + 22i$ are rest points.

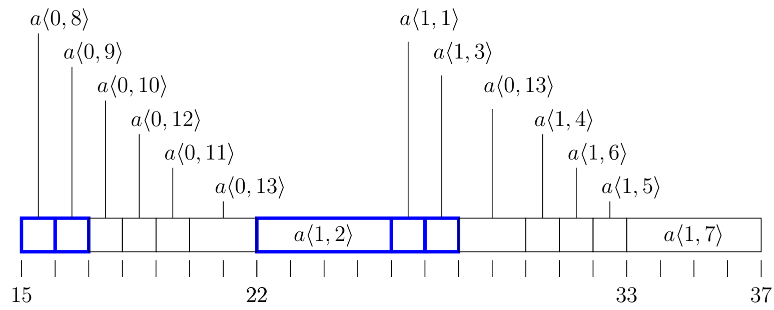

Figure 10: Feasible schedule for activities of Example 17 that are released between $i - \Pi$ and $i - 1$ ($i = 37$ is a rest point).

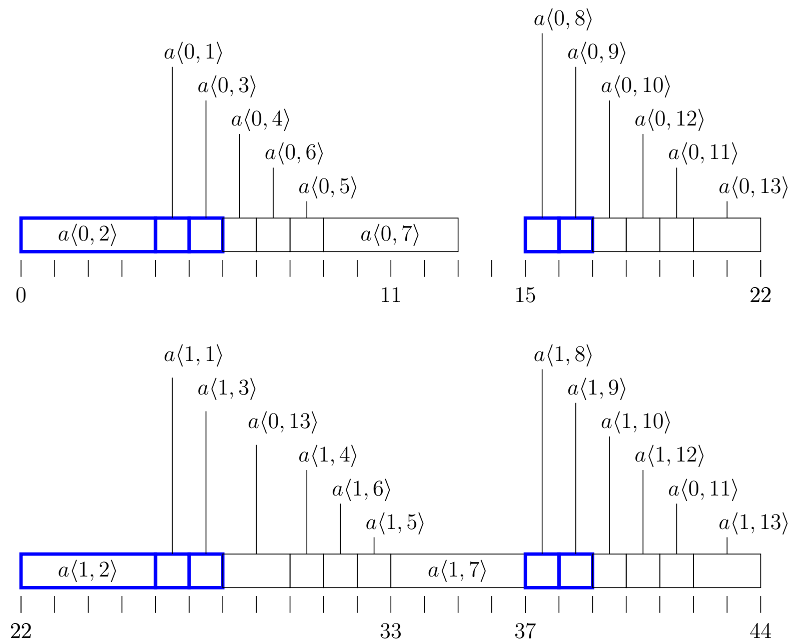

Figure 11: Prefix of a feasible schedule for Example 17.

```
sensor
  port s type int time 1
actuator
  port a type int init 0
output
  port o1 type int init 0
  port o2 type int init 0

task t1 input i1 output o1 function f1
task t2 input i2 output o2 function f2

driver d1 source s guard true destination i1 function h1
driver d2 source o1 guard true destination i2 function h2
driver d3 source o2 guard true destination a function h3 time 1

mode m period 10 ports o1, o2
  frequency 1 invoke t1 driver d1
  frequency 1 invoke t2 driver d2
  frequency 1 update d3

start m
```

Figure 12: An unconditional Giotto program.

4.1 Actuator- and sensor-dependent Giotto programs

We now define the class of single-mode Giotto programs for which we synthesize schedules. To begin with, we shall limit our attention to Giotto programs which are *unconditional*:

Definition 18 (unconditional program). A single-mode Giotto program is *unconditional* if the guard of every driver is *true*, i.e., for every driver $d \in \mathsf{Drivers}$ and every port valuation $v \in \mathsf{Vals}[\mathsf{Ports}]$, $\mathsf{g}[d](v) = \mathit{true}$. □

We will shortly define the subclass of unconditional programs that interests us. For unconditional Giotto programs, the result of every driver and task invocation needs to be obtained. Unconditional programs therefore represent the worst case for the scheduler. Figure 12 shows an unconditional Giotto program that serves as a running example in this section. This program has two tasks and one actuator, all of which are invoked with frequency 1. Task t_1 reads sensor s (via driver d_1). Task t_2 reads the output of task t_1 (via driver d_2). Actuator driver d_3 reads the output of t_2.

We now review elements of the presentation from [HHK03]. An execution E of a Giotto program is an infinite sequence $C_0, C_1, \cdots$ of *configurations*. Configuration C_i occurs at time τ_i, when actuators are updated, sensors are read, and task drivers and tasks are invoked. For a single-mode Giotto program, $\tau_i = i(\pi/\omega)$, where π is the period of the single mode m, and ω is the lcm of the frequencies of task invocations and actuator updates of m. An execution E gives rise to a set $\mathcal{J}_E$ of *jobs* $j[i,k]$. Here, the *job action* j is the job type; the first index i is the index of configuration C_i; and the second index k is the Giotto micro step at which the action is performed. Jobs of unconditional Giotto programs have four combinations of j and k:

- *Task jobs* are of the form $t[i, 1]$.
- *Actuator driver jobs* are of the form $\mathit{true}(d)[i, 2]$.
- *Sensor read jobs* are of the form $\mathit{read}(s)[i, 3]$.
- *Task driver jobs* are of the form $\mathit{true}(d)[i, 7]$.

Communication between jobs constrains the permissible order of their execution: if job $j'[i',k']$ reads a port last written by $j[i,k]$, then $j[i,k]$ *precedes* $j'[i',k']$ (in symbols, $j[i,k] \prec_E j'[i',k']$). Task jobs and task driver jobs may be executed at any time, subject to the constraints of $\prec_E$, and are thus called *floating jobs*. Actuator driver jobs $d[i,2]$ and sensor read jobs $s[i,3]$ must be executed close to time τ_i, and are thus called *fixed jobs*. For any executions E and E' of an unconditional program, $\mathcal{J}_E = \mathcal{J}_{E'}$ and $\prec_E = \prec_{E'}$. We thus write $\mathcal{J}$ and $\prec$ instead of $\mathcal{J}_E$ and $\prec_E$. Let $\prec^+$ denote the transitive closure of $\prec$. A *platform annotation* is a function wcet mapping each action j to a positive integer $\mathsf{wcet}(j)$, the worst-case execution time of j.

We now further focus on the class of single-mode programs of interest.

Definition 19 (actuator- and sensor-dependent program). A Giotto program is *actuator-dependent* if for every floating job $j[i,k]$, there exists a fixed job $j'[i',k']$ such that $j[i,k] \prec j'[i',k']$. A Giotto program is *sensor-dependent* if there exists $i^* \in \mathbb{Z}^{\geq 0}$ such that for every floating job $j[i,k] \in \mathcal{J}$ with $i \geq i^*$, there exists a fixed job $j'[i',k'] \in \mathcal{J}$ such that $j'[i',k'] \prec j[i,k]$. □

In an actuator-dependent program, every floating job precedes some fixed job.[9] In a sensor-dependent program, there is some configuration C_{i^*} after which every floating job is preceded by some fixed job.

Example 20. To illustrate the definition of actuator- and sensor-dependence, consider Figure 13, which shows a portion of the graph $(A, \prec)$ of the Giotto program of Figure 12. As the figure shows, every floating job precedes some fixed job; thus, the program of Figure 12 is actuator-dependent. Some floating jobs, for example $\mathit{true}(d_2)[0,7]$, are not preceded by a fixed job. However, every floating job $j[i,k]$ with $i \geq 2$ is preceded by a fixed job; thus, the program of Figure 12 is sensor-dependent. □

For sensor-dependent programs, a floating job not preceded by a fixed job can be computed prior to runtime; moreover, some floating jobs $j[i,k]$ may be computed considerably earlier than τ_i. Sensor- and actuator-dependent programs form an important class of Giotto programs, since Giotto is designed for applications that process sensor data and use the results to effect actuators. In the remainder of this report, all Giotto programs will be unconditional, sensor- and actuator-dependent.

4.2 A scheduling problem for Giotto

We now define the scheduling problem for which we will develop an algorithm. We are concerned with scheduling only those jobs in $\mathcal{J}$ that are preceded by a fixed job. We wish to schedule these jobs so that (1) the constraints of $\prec$ are respected, (2) every job $j[i,k]$ is scheduled for at least $\mathsf{wcet}(j)$ time units, and (3) for fixed jobs $j[i,k]$ the temporal difference between τ_i and the time at which $j[i,k]$ is scheduled is minimized. We now formalize these requirements.

Definition 21 (ε-feasibility). Let the set $\mathcal{J}^*$ be defined as follows: $j[i,k] \in \mathcal{J}^*$ if and only if $j[i,k] \in \mathcal{J}$, and either $j[i,k]$ is fixed or there exists a fixed job $j'[i',k']$ with $j'[i',k'] \prec^+ j[i,k]$. A Giotto program G is *ε-feasible* if there exists a schedule S such that for every job $j[i,k] \in \mathcal{J}^*$:

- The total execution time $\mathcal{T}_S(j[i,k])$ equals $\mathsf{wcet}(j)$.
- If $j[i,k]$ precedes some job $j'[i',k']$, then $\mathcal{F}_S(j[i,k]) \leq \mathcal{S}_S(j'[i',k'])$.
- If $j[i,k]$ is an actuator driver job, then $\tau_i - \varepsilon \leq \mathcal{S}_S(j[i,k])$ and $\mathcal{F}_S(j[i,k]) \leq \tau_i$.
- If $j[i,k]$ is a sensor read job, then $\tau_i \leq \mathcal{S}_S(j[i,k])$ and $\mathcal{F}_S(j[i,k]) \leq \tau_i + \varepsilon$. □

The quantity ε was termed *jitter tolerance* in [HHK03]. A large jitter tolerance is clearly undesirable. In particular, for jitter tolerances larger than π/ω, the sensor and actuator jobs of one configuration C_i can be executed at τ_{i-1} or τ_{i+1}, which is unacceptably early or late. This motivates the following problem statements:

Problem 22. Does a program G have an ε-feasible schedule for some $\varepsilon \leq \pi/\omega$?

[9] An weaker but equivalent definition of actuator-dependence is that a program is *actuator-dependent* if there exists $i^* \in \mathbb{Z}^{\geq 0}$ such that for every floating job $j[i,k]$ with $i \geq i^*$, there exists a fixed job $j'[i',k']$ such that $j[i,k] \prec j'[i',k']$. However, the weak and strong definitions for sensor-dependence are not equivalent.

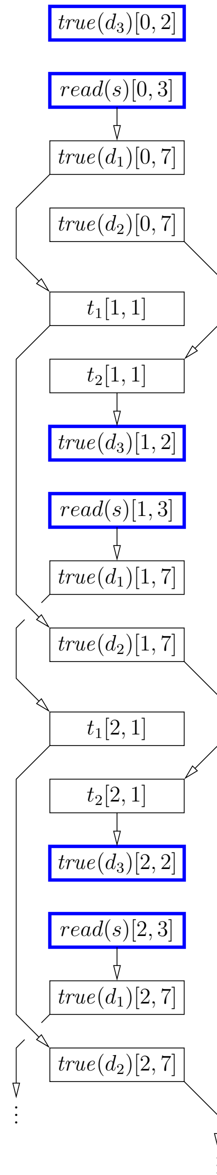

Figure 13: The dataflow graph $(A, \prec)$ of the Giotto program of Figure 12.

Problem 23. If so, what is the smallest ε^* such that G has an ε^*-feasible schedule?

Problem 24. Given this minimum ε^*, synthesize an ε^*-feasible schedule.

For a technical reason, we shall assume that actuator values, once written, are never read. The purpose of this assumption is to ensure that any ε-feasible schedule S can be transformed into another ε-feasible schedule S' in which all actuator drivers of configuration C_i are executed without interruption immediately prior to each time τ_i. This transformation is necessary to prove Theorem 31 below. We wish to emphasize that this assumption is merely technical; it may be removed by splitting each actuator driver into two parts, one that is purely functional (it computes and writes the actuator port value), and another that is purely operational (it reads the port value and interacts with the device).

4.3 The reduced dataflow graph

The remainder of this section develops an algorithm for these problems. Our approach is, given G and wcet, (1) to generate an instance $P[G, \mathsf{wcet}]$ of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$, and (2) to apply the algorithm of Section 3.3.2 to $P[G, \mathsf{wcet}]$. For (1), we must first determine which jobs are in $\mathcal{J}^*$, and we must partition $\mathcal{J}^*$ into $A_0, A_1, \cdots$. To this end, we introduce the *reduced dataflow graph*, which captures the time delays of an execution of the program G:[10]

Definition 25 (reduced dataflow graph). The *reduced dataflow graph* of an unconditional Giotto program is a edge-weighted directed graph (V, E, W), where the vertices V, edges $E \subseteq V \times V$, and weight function $W : E \to \mathbb{Z}^{\geq 0}$ are defined as follows:

- The set V is $\{j[i,k] \in A \mid i \in [0 \,..\, \omega - 1]\}$.
- The pair $e = (j[i,k], j'[i',k']) \in V \times V$ is in E if:
  - $j[i,k] \prec j'[i',k']$. In this case, we define $W(e) = i' - i$.
  - $j[i,k] \prec j'[i'+\omega,k']$. In this case, we define $W(e) = i' + \omega - i$. □

The reduced dataflow graph lets us determine an upper bound on the latest time at which a floating activity $j[i,k] \in \mathcal{J}^*$ may execute. Let $\ell = i \bmod \omega$. Let $L_{j[\ell,k]}$ be the minimum path length in the reduced dataflow graph from $j[\ell,k]$ to any fixed activity $j'[\ell',k'] \in V$. The earliest configuration that invokes a fixed transitive successor of $j[i,k]$ is $C_{i+L_{j[\ell,k]}}$. Thus:

Proposition 26. Let S be a ε-feasible schedule for *any* $\varepsilon > 0$. Then:

$$\mathcal{F}_S(j[i,k]) \leq (i + L_{j[\ell,k]})(\pi/\omega)$$

Similarly, the reduced dataflow graph lets us determine a lower bound on the earliest time at which a floating activity $j[i,k] \in \mathcal{J}^*$ may execute. Again let $\ell = i \bmod \omega$, and let $E_{j[\ell,k]}$ be the minimum path length in the reduced dataflow graph from any fixed activity $j'[\ell',k'] \in V$ to $j[\ell,k]$. The latest configuration that invokes a fixed transitive predecessor of $j[i,k]$ is $C_{i-E_{j[\ell,k]}}$. Thus:

Proposition 27. Let S be a ε-feasible schedule for *any* $\varepsilon > 0$. Then:

$$\mathcal{S}_S(j[i,k]) \geq (i - E_{j[\ell,k]})(\pi/\omega)$$

Finally, the reduced dataflow graph lets us determine the set $\mathcal{J}^*$ of jobs which cannot be computed prior to runtime. Consider a floating job $j[i,k]$: the latest fixed job which transitively precedes $j[i,k]$ has a configuration number of $i - E_{j[\ell,k]}$. If $i - E_{j[\ell,k]} < 0$, then no fixed job transitively precedes $j[i,k]$, and $j[i,k] \notin \mathcal{J}^*$. Thus:

Proposition 28. $\mathcal{J}^*$ is the union of the following two sets:

$$\{j[i,k] \in \mathcal{J} \mid j[i,k] \text{ is a fixed job}\}$$
$$\left\{j[i,k] \in \mathcal{J} \mid j[i,k] \text{ is a floating job and } i - E_{j[\ell,k]} \geq 0\right\}$$

[10]The concept of a reduced dataflow graph first appeared in [KMW67], where the delays were allowed to be multidimensional. [KMW67] shows that multidimensional delays necessitate the buffering of unboundedly much data as time progresses; fortunately, our delays are unidimensional. Reduced dataflow graphs are commonly used to study the parallelization of programming languages (cf. [DRV00]).

4.4 The scheduling problem generated by a Giotto program

We are now in a position to define the scheduling problem instance $P[G, \mathsf{wcet}]$ generated by a Giotto program G and execution times wcet:

Definition 29 (the scheduling problem generated by a Giotto program). $P[G, \mathsf{wcet}]$ is a tuple $(A, t, r, d, \ll, \Pi)$, defined as follows:

- For $\ell = 0, 1, \cdots$, let

$$\begin{array}{rcl} A_\ell & = & \{\text{fixed jobs } j[i,k] \in \mathcal{J} \mid i \in [\ell\omega \,..\, (\ell+1)\omega - 1]\} \\ & \cup & \big\{\text{floating jobs } j[i,k] \in \mathcal{J} \mid i - E_{j[i \bmod \omega, k]} \in [\ell\omega \,..\, (\ell+1)\omega - 1]\big\} \end{array}$$

  Let $A = \bigcup_{\ell=0}^{\infty} A_\ell$.

- For $j[i,k] \in A$, let $t(j[i,k]) = \mathsf{wcet}(j)$.

- Let Act_i be the actuator driver jobs with configuration number i, i.e., the set $\{d[i,2] \in \mathcal{J}\}$, and let Sense_i be the sensor read jobs with configuration number i, i.e., the set $\{s[i,3] \in \mathcal{J}\}$. For $j[i,k] \in \mathsf{Act}_i$,

$$\begin{array}{rcl} r(j[i,k]) & = & i(\Pi/\omega) - \sum_{d[i,2] \in \mathsf{Act}_i} \mathsf{wcet}(d) \\ d(j[i,k]) & = & i(\Pi/\omega) \end{array}$$

  For $j[i,k] \in \mathsf{Sense}_i$,

$$\begin{array}{rcl} r(j[i,k]) & = & i(\Pi/\omega) \\ d(j[i,k]) & = & i(\Pi/\omega) + \sum_{s[i,3] \in \mathsf{Sense}_i} \mathsf{wcet}(s) \end{array}$$

  Otherwise, $j[i,k]$ is a floating job, and

$$\begin{array}{rcl} r(j[i,k]) & = & (i - E_{j[i \bmod \omega, k]})(\Pi/\omega) \\ d(j[i,k]) & = & (i + L_{j[i \bmod \omega, k]})(\Pi/\omega) \end{array}$$

- The relation $\ll$ is defined as follows. Let $j[i,\ell], j'[i',\ell']$ be two members of A. Then

$$j[i,k] \ll j'[i',k'] \quad \text{iff} \quad j[i,k] \prec j'[i',k']$$

- Finally, Π is the period π of mode m. □

It may be verified that $P[G, \mathsf{wcet}]$ satisfies the conditions of Definition 1, with the exception of condition (2). We now investigate the extent to which (2) holds. Note that for $a \in A_0$,

$$r(a) \in \left[-\sum_{d[\omega,2] \in \mathsf{Act}_\omega} \mathsf{wcet}(d) \,..\, \pi - \pi/\omega \right] \tag{15}$$

Consider two adjacent configurations C_i and C_{i+1}, where $i \in [0 \,..\, \omega - 1]$. If it is not the case that

$$\sum_{s[i,3] \in \mathsf{Sense}_i} \mathsf{wcet}(s) + \sum_{d[i+1,2] \in \mathsf{Act}_{i+1}} \mathsf{wcet}(d) \leq \pi/\omega \tag{16}$$

then the program G cannot be ε-feasible for any $\varepsilon \leq \pi/\omega$. Whether (16) holds may be checked in time polynomial in $P[G, \mathsf{wcet}]$, by examining all activities in $A_0 \cup A_1$. Suppose on the other hand that (16) holds for all $i \in [0 \,..\, \omega - 1]$, and in particular for $i = \omega - 1$. If $\sum_{d[\omega,2] \in \mathsf{Act}_\omega} \mathsf{wcet}(d) = \pi/\omega$, then $\sum_{s[\omega-1,3] \in \mathsf{Sense}_{\omega-1}} \mathsf{wcet}(s) = 0$. In this case, $\mathsf{Sense}_{\omega-1} = \emptyset$, so that the upper bound of (15) is strict, i.e., for $a \in A_0$,

$$\begin{array}{rcl} r(a) & < & \pi - \pi/\omega \\ & = & \pi - \sum_{d[\omega,2] \in \mathsf{Act}_\omega} \mathsf{wcet}(d) \end{array}$$

If $\sum_{d[\omega,2]\in \mathsf{Act}_\omega} \mathsf{wcet}(d) < \pi/\omega$, then by (15), for $a \in A_0$,

$$\begin{array}{rcl} r(a) & \leq & \pi - \pi/\omega \\ & < & \pi - \displaystyle\sum_{d[\omega,2]\in \mathsf{Act}_\omega} \mathsf{wcet}(d) \end{array}$$

We have established the following proposition:

Proposition 30. Whether (16) holds may be checked in time polynomial in $P[G, \mathsf{wcet}]$. If (16) does not hold, then G is not ε-feasible for any $\varepsilon \leq \pi/\omega$. If (16) holds, then

$$r(a) \in \left[-\sum_{d[\omega,2]\in \mathsf{Act}_\omega} \mathsf{wcet}(d) \;..\; \pi - 1 - \sum_{d[\omega,2]\in \mathsf{Act}_\omega} \mathsf{wcet}(d) \right] \tag{17}$$

The fact that the release times of $P[G, \mathsf{wcet}]$ satisfy the modified condition (17) presents no difficulties for the scheduling algorithm of Section 3.3.2. Suppose that (16) holds, and let S denote the schedule obtained by running the algorithm of Section 3.3.2 on input $P[G, \mathsf{wcet}]$. Suppose there exists an ε'-feasible schedule S' for some $\varepsilon' \leq \pi/\omega$. Using an exchange argument, it may be shown that S' can be transformed into S, and that S is ε-feasible for some $\varepsilon \leq \varepsilon'$. Further, it may be verified that S is feasible. Thus, if G has an ε'-feasible schedule for some $\varepsilon' \leq \pi/\omega$, then S is feasible. The converse also holds: by the construction of $P[G, \mathsf{wcet}]$, if S is feasible, then S is ε-feasible for some $\varepsilon \leq \pi/\omega$. Note that all sensors and actuators drivers are executed at configuration C_0. Thus, the maximum jitter in S occurs at C_0, i.e., $\varepsilon^* = \max\left\{\sum_{a\in \mathsf{Act}_0} \mathsf{wcet}(a), \sum_{a\in \mathsf{Sense}_0} \mathsf{wcet}(a)\right\}$. Finally, since the jitter tolerance ε attained by S is at most the jitter tolerance ε' obtained by an arbitrary schedule S', ε is the minimum jitter tolerance ε^*, and S is a ε^*-feasible schedule. We have established the following:

Theorem 31. Problems 22, 23, and 24 may be solved as follows:

1. G has an ε-feasible schedule for some $\varepsilon \leq \pi/\omega$ if and only if S is feasible.
2. If S is feasible, then $\varepsilon^* = \max\left\{\sum_{a\in \mathsf{Act}_0} \mathsf{wcet}(a), \sum_{a\in \mathsf{Sense}_0} \mathsf{wcet}(a)\right\}$.
3. If S is feasible, then S is an ε^*-feasible schedule.

The problem instance $P[G, \mathsf{wcet}]$ may be generated in time pseudopolynomial in the frequencies of the task invocations and actuator updates of mode m. Since the algorithm of Section 3.3.2 is polynomial time, we conclude that Problems 22, 23, and 24 may be solved in pseudopolynomial time. Whether a fully polynomial-time algorithm for these problems exists is an open question.

5 Additional optimizations

We now describe two ways to optimize a runtime system that executes an instance of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$. The first optimization (Section 5.1) is to aggregate distinct activities into the same thread. This reduces the number of threads, and consequently the memory footprint and the context switch overhead. The second optimization (Section 5.2) is to use a single stack to execute activities. This makes a context switch not much more expensive than a function call [Wir96].

5.1 Activity aggregation

Consider an instance P of $1 \mid r_j;\, d_j;\, prec;\, pmtn;\, period \mid -$, and the set of activities

$$A^r_d = \{a \in A \mid r^*(a) = r \text{ and } d^*(a) = d\}$$

At runtime, the activities in A^r_d are executed in some number of threads. How many threads are necessary? We claim that a single thread T^r_d suffices. T^r_d executes the activities in A^r_d in any linear order consistent

with the partial order $\ll$. The scheduler uses an earliest deadline first policy to schedule threads, where the deadline of T_d^r is d. Further, the scheduler resolves ties between threads with the same deadline in favor of the thread T_d^r with minimum release time r. The question here is, given that each activity a may run for *less* than its worst-case time $t(a)$ to run, can some precedence constraint is violated? To answer this question, consider two activities a_1 and a_2 such that $a_1 \ll a_2$. If a_1 and a_2 are in the same set A_d^r, then a_1 will finish before a_2 begins, since T_d^r linearizes $\ll$. Suppose that a_1 and a_2 are in distinct sets $A_{d_1}^{r_1} \neq A_{d_2}^{r_2}$, respectively. By the definition of r^* and d^*, since $a_1 \ll a_2$, $r_1 \leq r_2$ and $d_1 \leq d_2$. There are two cases to consider:

1. If $d_1 < d_2$, then $T_{d_1}^{r_1}$ will execute in preference to $T_{d_2}^{r_2}$. Thus a_1 will finish before a_2 begins.
2. If $d_1 = d_2$, since $A_{d_1}^{r_1} \neq A_{d_2}^{r_2}$, $r_1 < r_2$. Again in this case, $T_{d_1}^{r_1}$ will execute in preference to $T_{d_2}^{r_2}$.

It follows that no precedence constraint can be violated.

In the example of Figures 12 and 13, the sets A_d^r are:

$$
\begin{array}{lcl}
A_0^{-1} & = & \{\mathit{true}(d_3)[0,2]\} \\
A_1^{0} & = & \{\mathit{read}(s)[0,3]\} \\
A_{20}^{0} & = & \{\mathit{true}(d_1)[0,7],\ \ t_1[1,1],\ \ \mathit{true}(d_2)[1,7],\ \ t_2[2,1]\} \\
A_{10}^{9} & = & \{\mathit{true}(d_3)[1,2]\} \\
A_{11}^{10} & = & \{\mathit{read}(s)[1,3]\} \\
A_{30}^{10} & = & \{\mathit{true}(d_1)[1,7],\ \ t_1[2,1],\ \ \mathit{true}(d_2)[2,7],\ \ t_2[3,1]\} \\
\ldots & &
\end{array}
$$

The jobs $\mathit{true}(d_2)[0,7]$ and $t_2[1,1]$ appear neither in A nor in A_d^r, since these jobs may be computed before runtime.

5.2 Single-stack implementation

Definition 32 (balanced schedule). Let S be a schedule, as defined by Definition 2 or Definition 4. We say that S is *balanced* if for any two activities $a_1, a_2 \in A$, it is not the case that $\mathcal{S}_S(a_1) < \mathcal{S}_S(a_2) < \mathcal{F}_S(a_1) < \mathcal{F}_S(a_2)$. □

Such schedules are called balanced because they correspond to strings in a balanced parenthesis language, where each opening parenthesis $(_a$ denotes the start of activity a, and each closing parenthesis $)_a$ denotes the completion of a. EDF$^{\prec}$ and the algorithm of Section 3.3 always produce balanced schedules, since if activity a_1 preempts activity a_2, then a_1 has an earlier deadline than a_2, and a_1 will complete before a_2.[11] These properties hold not only for the schedules produced by EDF-based algorithms, but for balanced schedules in general. The following proposition, which may be proved using an exchange argument, makes this notion precise.

Proposition 33. Any balanced schedule $S = (I, e)$ may be transformed into a balanced schedule $S' = (I', e')$ with the following properties:

1. For any activity a in the range of e,

$$
\begin{array}{rcl}
\mathcal{T}_S(a) & = & \mathcal{T}_{S'}(a) \\
\mathcal{S}_S(a) & \leq & \mathcal{S}_{S'}(a) \\
\mathcal{F}_{S'}(a) & \leq & \mathcal{F}_S(a)
\end{array}
$$

   Thus, if S satisfies an instance P of $1 \mid r_j; d_j; \mathit{prec}; \mathit{pmtn}; \mathit{period} \mid -$, then S' also satisfies P.

2. Let a_1 and a_2 be activities in the range of e such that

$$\mathcal{S}_{S'}(a_1) < \mathcal{S}_{S'}(a_2) < \mathcal{F}_{S'}(a_2) < \mathcal{F}_{S'}(a_1)$$

   Then $e'(i') \neq a_1$ for any interval $i' \in I'$ such that $i' \cap [\mathcal{S}_{S'}(a_2), \mathcal{F}_{S'}(a_2)] \neq \emptyset$. In other words, if a_2 preempts a_1, then a_2 finishes before a_1 executes again.

[11] The rate monotonic scheduling algorithm [LL73] also produces balanced schedules.

Balanced schedules are attractive for two additional reasons. First, in a balanced schedule the overhead due to context switches may be bounded: each thread gets charged one context switch when its starts, and one when it finishes. For finite schedules this bound is $2tc$, where t is the number of threads and c is the time required for a context switch. This observation is originally due to A.K. Mok and M.L. Dertouzos [MD78] for schedules produced by EDF-based algorithms. Second, balanced schedules may be implemented using a single stack. In most current programming language implementations, each thread uses a pushdown stack to execute function calls. Operating systems, including most real-time operating systems, typically use separate stacks for distinct threads. For balanced schedules, the same stack space may be used by different threads, as we will explain below. This has the advantage of saving time during context switches: as we will see, preemption becomes no more expensive than a jump into an interrupt service routine, plus an invocation of the (minimal) runtime scheduler, plus a function call. Storing and restoring additional per-thread information is no longer necessary. The observation that a single stack suffices for fixed-priority systems is due to N. Wirth [Wir96]. The generalization to arbitrary balanced schedules is, we believe, original.

We now explain how to share a single stack in a balanced schedule. We focus on time instants when the runtime scheduler is active: (1) thread startup ($\mathcal{S}_S(a)$ for some activity a), and (2) thread termination ($\mathcal{F}_S(a)$).[12] At thread startup, the runtime scheduler receives control; this is frequently accomplished using a timer interrupt. The stack is in the state in which the interrupted thread has left it. The runtime scheduler places the program counter of the interrupted thread on the stack. The runtime scheduler then jumps to the start address of the thread being started. The new thread runs for some time, perhaps getting preempted by other threads. We assume that the new thread leaves the stack in the same state in which it found it, i.e., with the program counter of the preempted thread on top. At thread termination, the runtime scheduler pops and returns to the program counter of the interrupted thread. Because the schedule is balanced, the interrupted thread need not execute (and thus use the stack) between the start time and the finish time of the interrupting thread.

6 Conclusion

This report presented an approach to scheduling single-mode Giotto programs for a single processor. To account for the infinite nature of Giotto programs, we extended the classical scheduling problem $1 \mid r_j;\ d_j;\ prec;\ pmtn \mid -$ to a periodic version. We developed an optimal algorithm for this extended problem, based on the concept of rest points. We then showed how to translate a class of single-processor, single-mode Giotto programs into instances of this extended problem. This resulted in a pseudopolynomial-time scheduling algorithm for our class of programs.

References

[B+01] J. Błażewicz et al. *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, 2nd edition, 2001.

[BHR93] S.K. Baruah, R.R. Howell, and L.E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[Bła76] J. Błażewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proc. Intl. Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65. North-Holland, 1976.

[Bru01] P. Brucker. *Scheduling Algorithms.* Springer-Verlag, 2001.

[BS93] G.C. Buttazzo and J.A. Stankovic. RED: Robust earliest deadline first scheduling. In *Proc. 3rd Intl. Workshop on Responsive Computer Systems*, pages 100–111. IEEE, 1993.

[12] The scheduler may also be active if a thread terminates before its scheduled finish time. This commonly occurs if the actual execution time of the thread is less than its worst-case execution time. At such times, the scheduler may execute soft real-time threads, using separate stacks if necessary. Here we discuss the use of a single stack only for hard real-time threads.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.

[HHK03] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[HKMM02] T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. 2nd Intl. Workshop on Embedded Software*, LNCS 2491, pages 76–92. Springer-Verlag, 2002.

[HLv97] J.A. Hoogeveen, J.K. Lenstra, and S.L. van de Velde. Sequencing and scheduling. In *Annotated Bibliographies in Combinatorial Optimization*, pages 181–197. Wiley, 1997.

[Kim94] Y.S. Kim. An optimal scheduling algorithm for preemptable real-time tasks. *Information Processing Letters*, 50(1):43–48, 1994.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

[Kop97] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Kluwer, 1997.

[KSHP02] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *Proc. 2nd Intl. Workshop on Embedded Software*, LNCS 2491, pages 46–60, 2002.

[Len77] J.K. Lenstra. Sequencing by enumerative methods. Technical Report 69, Mathematisch Centrum, Amsterdam, 1977.

[LL73] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[LLK82] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: a survey. In *Deterministic and Stochastic Scheduling*, pages 35–73. Reidel, 1982.

[LLLK82] B.J. Lageweg, J.K. Lenstra, E.L. Lawler, and A.H.G Rinnooy Kan. Computer-aided complexity classification of combinatorial problems. *Communications of the ACM*, 25(11):817–822, 1982.

[MD78] A.K. Mok and M.L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. 7th Texas Conference on Computing Systems*, pages 5.1–5.12, 1978.

[Pin95] E. Pinson. The job shop scheduling problem: A concise survey and some recent developments. In *Scheduling Theory and its Applications*, pages 276–293. Wiley, 1995.

[SBS95] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proc. 16th IEEE Real-Time Systems Symposium*, pages 210–219. IEEE, 1995.

[Wir96] N. Wirth. Tasks versus threads: An alternative multiprocessing paradigm. *Software: Concepts and Tools*, 17:6–12, 1996.