In Proc. of 14th IEEE Real-Time Systems Symposium, pp. 232-242, December, 1993

Semantics-Based Compiler Transformations for Enhanced Schedulability *

Richard Gerber and Seongsoo Hong
Department of Computer Science
University of Maryland
College Park, MD 20742
rich@cs.umd.edu sshong@cs.umd.edu

Abstract

We present TCEL (Time-Constrained Event Language), whose timing semantics is based solely on the constrained relationships between observable events. Using this semantics, the unobservable code can be automatically moved to convert an unschedulable task set into a schedulable one. We illustrate this by an application of program-slicing, which we use to automatically tune control-domain systems driven by rate-monotonic scheduling.

1 Introduction

The construction of a real-time system involves reconciling two "adversersial" factors: the system's *real-time specification* and the *timing characteristics* of its hardware platform. Specification-based properties "come from above," and establish constraints between *occurrences of events* [3, 10]. An example is *the robot arm must receive a next-position update every 10 ms.* Such a constraint arises from the system's requirements, or from a detailed analysis of the application environment. On the other hand, the hardware platform's timing characteristics "come from below," and are affected by factors such as the CPU's architecture, instruction-cycle times, memory-cycle times, bus arbitration delays, etc.

The gap between a specification and its implementation platform frequently results in the painful process of system-tuning to achieve schedulability. This may involve re-implementing key modules in assembly language (or even in silicon), hand-optimizing the code, manually counting instruction-cycle times, and experimenting with various orderings of operations. As a last resort, entire subsystems may have to be re-designed altogether.

The goal of our research is to assist programmers in bridging this gap. Our approach consists of two interrelated factors: a programming language and compiler transformations. The real-time programming language is called TCEL (Time-Constrained Event Language), which contains first-class timing constructs, and whose semantics is based on time-constrained relationships between observable events. As the only timing constraints are imposed by observable events, the unobservable code can be transformed to automatically assist in the low level tuning process. As we show in this paper, it is precisely the TCEL semantics which makes the compiler transformations possible.

The TCEL Language. TCEL contains constructs quite similar to those developed in other experimental languages [9, 11, 13, 15, 18, 26]. In these approaches, however, timing constraints are established between *blocks of code*. The TCEL semantics, on the other hand, establishes constraints between the *observable events* within the code.

For example, consider a construct such as "**every** 10ms **do** B," where the block of code "B" is executed once every 10 milliseconds. In a code-based semantics, *all* of the code in B must fit properly within each 10 ms time-frame. In the TCEL event-based semantics, *only the observable events* in B must fit properly within the time-frame. This looser semantics yields two immediate benefits. First, the decoupling of timing constraints from code blocks enables a more straightforward implementation of an event-based specification. But more importantly, the unobservable code can be moved to automatically tune the program to its hardware environment.

In the sequel we consider all "**send**" and "**re-**

*This research is supported in part by NSF grant CCR-9209333 and ARPA contract N00014-91-C-0195.

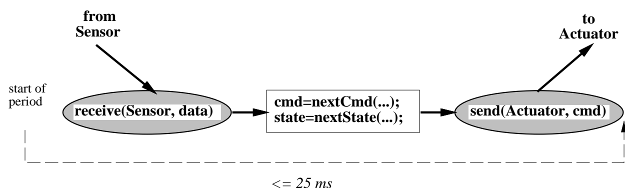


Figure 1: Structure of Controller Subsystem.

ceive" operations to be observable. Consider a simple real-time program specification, rendered pictorially in Figure 1. The requirements specification is also given below:

(1) Every 25ms, an external sensor sends a message to the controller, containing physical world measurements.
(2) The controller must receive every message.
(3) Using the sensor data and the current state, the controller computes a next-position command and sends it to an actuator.
(4) The command must be sent within each period.
(5) Based on the sensor-input, the controller updates its current state.

The TCEL program fragment below realizes the specification.

```
A1: every 25ms
    {
A2: receive(Sensor, data);
A3: cmd = nextCmd(state, data);
A4: state = nextState(state, data);
A5: send(Actuator, cmd);
    }
```

The system's only observable events are triggered instantaneously during the executions of the "**send**" and "**receive**" operations. The "**every**" statement establishes timing constraints *only* between these two operations. On the other hand, the local statements A3 and A4 are simply constrained by the program's natural control and data dependences.

Under a code-based semantics the program is interpreted in a different way; that is, the statements A2-A5 must be executed within a single frame. This interpretation is in fact *much* stronger than the requirements mandate, and indeed, may result in an unnecessary fault. For example, if the system experiences a transient overload caused by higher-priority tasks, the program may not meet its deadline.

In this case there are obvious remedies, which *would have to be performed by hand.* For example, part or all of the next-state update in A4 could be relocated beyond A5. Then, in the case of transient overload, this operation could be postponed beyond the end of the period. However, the necessary corrections would include manually decomposing A4, moving part of it, and adding necessary hooks for the scheduler to postpone a deadline. The actual changes would heavily depend on the particular characteristics of the computer, and thus, the very reason for using high-level timing constructs would be defeated.

Transforming Tasks for Enhanced Schedulability. The event-based semantics provides a foundation to automatically tune a real-time system. Returning to our example, a compiler transformation can be used to automatically decompose A4. Yet another transformation can relocate as much (or as little) code as is necessary to tolerate single-period overloads. In performing these transformations, the TCEL compiler uses the observable events as "semantic markers," which establish boundaries of code decomposition, and constrain the places where code can be moved.

In previous work we show how to use code-motion optimizations to resolve conflicts within single tasks [7]. These conflicts can arise when tasks have nested constraints; e.g., when deadlines are tighter than periods, or when there are inserted delay statements. Since these timing constraints may conflict with the task's execution time, it may appear to be inherently unschedulable. Hence the objective is to automatically achieve "internal" consistency between real-time requirements and elapsed execution time. Our approach is to use instruction-scheduling techniques [5], with which our compiler moves code from blocks constrained by tight deadlines into blocks with sufficient slack.

In this paper we address a more aggressive goal – inter-task transformations for schedulability. To help attack this problem, we have developed a compiler decomposition technique to support control-domain programs under rate-monotonic scheduling [16]. In particular, our technique can be viewed as a safe, automatic way to use *deadline postponement* [21]. (We explain deadline postponement in Section 3.3.) The framework consists of the following ingredients:

(1) An algorithm which uses standard rate-monotonic analysis to find unschedulable tasks, and determines the amount that they must be transformed.
(2) A *program slicer*, which decomposes a task and isolates the component that can have its deadline postponed.
(3) An online, dynamic adaptation to the rate-monotonic scheduler, which enforces precedence constraints between task iterations. (We call this

adaptation *priority exchange*.)

The remainder of the paper is organized as follows. In Section 2 we present an overview of the TCEL language. In Section 3 we motivate our transformation algorithm via a discussion of rate monotonic scheduling for control systems. We also present the algorithm itself. In Section 4 we provide a technical treatment of program slicing, and its applicability to task decompositions. We conclude in Section 5 with a discussion of related work in compilation techniques for real-time systems. We end the section (and the paper) with remarks on our future work.

2 Overview of TCEL

In this section we present two of TCEL's constructs to denote timing constraints within a program. Both constructs are syntactic descendents of the *temporal scope*, introduced in [13]. However, as we have stated, our semantics is quite different, in that it relies on constrained relationships between observable events.

We use the "**do**" construct to denote a sporadic program with relative timing constraints:

do
  ⟨reference block⟩
[**start after** t_{min}] [**start before** t_{max1}]
[**finish within** t_{max2}]
  ⟨constraint block⟩

The reference block (RB) and the constraint block (CB) are simply C statements, or alternatively, timing constructs themselves. The "**do**" construct induces the following timing constraints:

- **start after** t_{min}: There is a minimum delay of t_{min} between the last event executed in the RB, and the first event executed in the CB.
- **start before** t_{max1}: There is a maximum delay of t_{max1} between the last event executed in the RB, and the first event executed in the CB.
- **finish within** t_{max2}: There is a maximum delay of t_{max2} between the last event executed in the RB, and the last event executed in the CB.

Since either block may contain conditionals, depending on the program's state there may be several such events executed either "first" or "last."

The second real-time construct denotes a statement

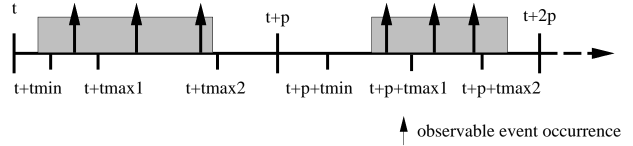


Figure 2: Behavior of Periodic Timing Construct.

with cyclic behavior of a positive periodicity:

every p [**while** ⟨condition⟩]
[**start after** t_{min}] [**start before** t_{max1}]
[**finish within** t_{max2}]
  ⟨constraint block⟩

As long as the "**while**" condition is true, the observable events in the constraint block execute every p time units. Akin to an untimed **while**-loop, when the condition evaluates to false the statement terminates. In its real-time behavior, the interpretation of the "**every**" construct is similar to that of "**do**." For example, assume that the statement is first scheduled at time t, and that the "**while**" condition is true for periods 0 through i. As depicted in Figure 2, the following constraints on events are induced for period i:

- **start after** t_{min}: The first event executed in the CB occurs after $t + ip + t_{min}$.
- **start before** t_{max1}: The first event executed in the CB occurs before $t + ip + t_{max1}$.
- **finish within** t_{max2}: The last event executed in the CB occurs before $t + ip + t_{max2}$.

Timing constructs may be arbitrarily nested. For example, consider the arm controller program specification in Figure 3 (left), and its TCEL realization in Figure 3 (right). From the event-based semantics, the timing constraints only apply to the observable events associated with the "**send**" and "**receive**" instructions. The local statements "`z1 = newcmd(dim, loc1)`" and "`z2 = newcmd(dim, loc2)`" can "float," and are constrained only by their natural dependences – as in any other program.

3 Scheduling with Compiler Transformations

The event-based semantics of TCEL gives a clear separation between constraints based on time, and those based on data and control dependences. Using the semantics as a foundation, we present a systematic task transformation algorithm, which is guided by the rate-monotonic scheduling theory. The algorithm is specifically intended for periodic tasks in the

1. Every 10.0 ms, receive message from sensor.
2. Delay at least 1.5 ms after receiving message.
3. If object was detected, send new commands to arm1 and arm2 within 4.0 ms of receiving message from sensor, and before next message.
	every 10.0 ms
	do
	receive(Sensor, dim);
	start after 1.5 ms **finish within** 4 ms
	if (!null(dim)) {
	z1 = newcmd(dim, loc1);
	z2 = newcmd(dim, loc2);
	send(arm1, z1);
	send(arm2, z2);
	}

Figure 3: The Specification (left), and The Program (right).

domain of guidance, navigation and control (GN&C). To motivate our transformation, we give an example set of GN&C tasks, which is shown to be unschedulable via a rate-monotonic scheduler. In addition, we discuss a simple (dynamic) modification of rate-monotonic scheduling algorithm to support our task transformation.

The objective of the algorithm is to enhance schedulability by transforming unschedulable tasks into multiple threads. Our method is to reduce CPU demands in overloaded frames. In this section we present the overview of the transformation algorithm, while in the next section we provide a more detailed development of program-slicing.

3.1 Characterization of Control Software

One of the major properties of control algorithms is that computations fit a fixed-rate algorithm paradigm [12]. Fixed-rate algorithms are those which execute repetitively with fixed periods. During each period, the physical world measurement data is sampled, and then actuator commands are computed. Meanwhile, a set of states is updated, based on current states and sampled data.

The dynamic behavior of GN&C systems can often be expressed by the following equations.

$$O_k = g(X_k, I_k) \qquad (1)$$

$$X_{k+1} = h(X_k, I_k) \qquad (2)$$

In these equations, I_k, X_k, and O_k respectively represent the input, current state, and output of the kth period, while g is an output generation function and h is a state evolution function.

Since GN&C equations are thought of as simultaneous relationships (not as a computation procedure), there are usually many valid computational orderings. One possible ordering of Eq 1 and 2 is given pictorially in Figure 4. It delineates the kth instance

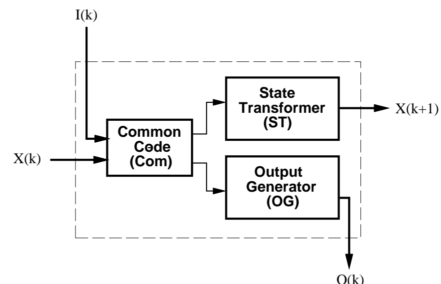


Figure 4: Task Decomposition in the kth Period.

of the control task. Note that in addition to two separate code components (the output generator `OG` and the state transformer `ST`), the common computational part is factored out, which then becomes the third code component `Com`. Thus function $g()$ corresponds to code components "`Com;OG`" while function $h()$ corresponds to "`Com;ST`".[1] In this figure arrows between code components represent precedences (denoted by "$\prec$") caused by data availability. For example, $\texttt{Com(k)} \prec \texttt{ST(k)}$ means that the kth instance of `Com` must be executed before the kth instance of `ST`.

In addition to these intra-task precedence constraints, the following orderings between components of task instances must be observed during the execution of the task.

(1) $\texttt{Com(k)};\texttt{ST(k)} \prec \texttt{Com(k+1)};\texttt{ST(k+1)}$
(2) $\texttt{Com(k)};\texttt{ST(k)} \prec \texttt{Com(k+1)};\texttt{OG(k+1)}$

3.2 Rate-Monotonic Schedulability Analysis

Rate-monotonic scheduling is well suited for control domain applications, not only because they possess the fixed-rate property, but also because an efficient schedulability test can be applied. Whereas

[1] The operator ";" denotes sequential composition of code fragments.

there has been significant amount of research on analytical characterization of the algorithm [14] and its practical application to various problems [21], there has been little work on the automatic translation of unschedulable task sets into schedulable ones. However, complex real-time system development can be supported by such an automatic scheme. For example, consider a task set which is derived from a real-time application. The derivation process involves careful implementation and tedious tuning of each task to satisfy various timing and functional constraints. In this setting, most of the task tuning must be repeated if the task set is determined to be unschedulable. Of course, the situation is even worse if the code is rehosted.

Our task transformation approach is motivated by the exact (necessary and sufficient) schedulability test in [14], which is based on the *critical instant* analysis. A critical instant for a task occurs whenever the task is initiated simultaneously with all higher priority tasks [16]. Let T_i and C_i be the period and the worst case computation time of task τ_i, respectively. Assume that the τ_i's are numbered in the increasing order of their T_i's. Since the rate-monotonic scheduling algorithm assigns a higher priority to a task with a smaller period, a task with a smaller number has a higher priority.

To determine if task τ_k can meet its deadline under the worst case phasing, it is necessary to check if there is point t in time in the interval $[0, T_k]$ (i.e. τ_k's critical instant), which satisfies the following inequality.

$$\sum_{i=1}^{k} \frac{C_i \lceil \frac{t}{T_i} \rceil}{t} \leq 1 \qquad (3)$$

To do so, we need to check only those points in the interval $[0, T_k]$ which are multiples of the periods of k tasks $\{\tau_1, \tau_2, \ldots, \tau_k\}$. They are called *scheduling points*, and become points where the left-hand-side of the above inequality achieves the local minima.

Example 1: Consider the case of three periodic tasks, where $U_i = C_i/T_i$.

$$\text{Task}(\tau_1) : C_1 = 4.0; T_1 = 10; U_1 = 0.4$$
$$\text{Task}(\tau_2) : C_2 = 4.0; T_2 = 16; U_2 = 0.25$$
$$\text{Task}(\tau_3) : C_3 = 6.41; T_3 = 25; U_3 = 0.2612$$

The source code of task τ_3 is given in Figure 5, where the bracketed numbers represent the best and worst-case execution times, respectively. These times are generated by a timing analysis tool, such as those found in [20, 27].

The two highest priority tasks τ_1 and τ_2 are schedulable, because their total utilization factor ($U_1 + U_2 = 0.65$) is below the rate-monotonic utilization bound:

```
      every 25ms
      {
L1:   receive(Sensor, data);              [0.2ms,0.5ms]
L2:   if (!null(data))                    [0.05ms,0.06ms]
        {
L3:     t1 = F1(state);                   [0.8ms,1.05ms]
L4:     t2 = F2(state);                   [0.9ms,1.35ms]
L5:     t3 = F3(data);                    [0.9ms,1.35ms]
L6:     t4 = F4(data);                    [0.9ms,1.35ms]
L7:     cmd = t1 * ( t3 + t4 );           [0.09ms,0.1ms]
L8:     send(Actuator, cmd);              [0.2ms,0.5ms]
L9:     state = t1 * ( t2 + t3 );         [0.11ms,0.15ms]
        }
L10:  }
```

Figure 5: TCEL Program for Task τ_3.

$0.83 = 2(2^{1/2} - 1)$ [16]. On the other hand, the entire task set is not schedulable, because the total utilization factor exceeds 1 at all scheduling points within the critical instant of τ_3 as shown below:

$$C_1 + C_2 + C_3 \leq T_1 \qquad (4 + 4 + 6.41 > 10)$$
$$2C_1 + C_2 + C_3 \leq T_2 \qquad (8 + 4 + 6.41 > 16)$$
$$2C_1 + 2C_2 + C_3 \leq 2T_1 \qquad (8 + 8 + 6.41 > 20)$$
$$3C_1 + 2C_2 + C_3 \leq T_3 \qquad (12 + 8 + 6.41 > 25)$$

□

The reason the above task set is unschedulable is obvious: the computation demands exceed all available time at τ_3's critical instant.

3.3 Task Transformation Algorithm

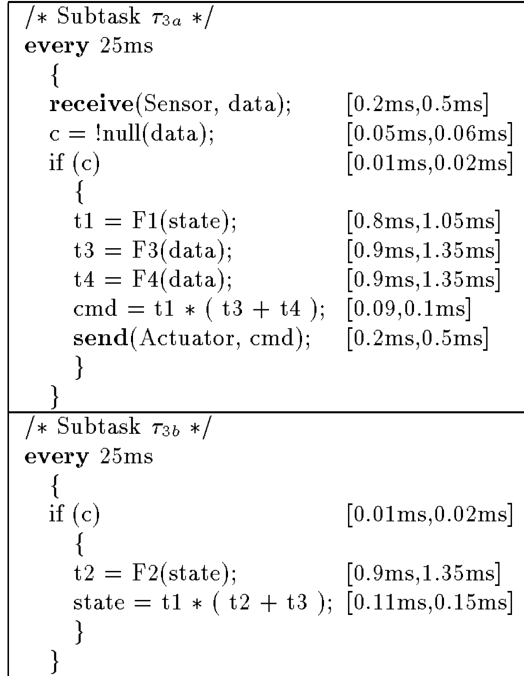
A straightforward technique to achieve schedulability is to let some of τ_3's code "slide" into the next period. This can be done by postponing the deadline (i.e. expanding the period) of τ_3, as suggested by Sha et al. in [21]. The application of deadline postponement to task τ can be described with the following steps:

Step 1 Task τ is duplicated into two tasks τ_x and τ_y.

Step 2 Both τ_x and τ_y are given $2T$ as their period, where T is τ's original period.

Step 3 τ_x is initiated at times $0, 2T, \ldots$, while τ_y is initiated at times $T, 3T, \ldots$.

A task transformation is *safe* only when the resultant task preserves the original timing and functional semantics. Deadline postponement is, however, not a safe transformation, since its correctness depends on the timing constraints of applications. Particularly it may not be applicable to GN&C domain software,



```
/* Subtask τ3a */
every 25ms
  {
  receive(Sensor, data);              [0.2ms,0.5ms]
  c = !null(data);                    [0.05ms,0.06ms]
  if (c)                              [0.01ms,0.02ms]
    {
    t1 = F1(state);                   [0.8ms,1.05ms]
    t3 = F3(data);                    [0.9ms,1.35ms]
    t4 = F4(data);                    [0.9ms,1.35ms]
    cmd = t1 * ( t3 + t4 );           [0.09,0.1ms]
    send(Actuator, cmd);              [0.2ms,0.5ms]
    }
  }
/* Subtask τ3b */
every 25ms
  {
  if (c)                              [0.01ms,0.02ms]
    {
    t2 = F2(state);                   [0.9ms,1.35ms]
    state = t1 * ( t2 + t3 );         [0.11ms,0.15ms]
    }
  }
```

Figure 6: Two Decomposed Subtasks.

in which the computation of (observable) output operations has a strict deadline and depends on current state and input. The problem with deadline postponement is that some observable events may miss their deadlines.

We solve this problem via a *compiler-driven* task decomposition technique. We put the greatest emphasis on preserving the timing behavior of observable events and the precedence constraints derived in the Section 3.1. Before presenting a systematic procedure, we describe our approach with the task set used in Example 1.

First, we decompose τ_3's code into two parts: (1) code that computes the output command (denoted by τ_{3a}) and (2) code that computes the state update (denoted by τ_{3b}). Figure 6 shows the subtasks which result from the decomposition. Note that subtasks τ_{3a} and τ_{3b} correspond to "`Com;OG`" and "`ST`" respectively in Figure 4. Their computation times are calculated by adding the individual execution times of the statements of the subtasks:

$$C_{3a} = 4.93\text{ms}, \quad C_{3b} = 1.52\text{ms}$$

Now that subtask τ_{3b} consists of only local computations, we can subject it to deadline postponement, and obtain two duplicated tasks, τ_{3b1} and τ_{3b2}. Of course, both of them have $2T_3$ as their period, and τ_{3b2} is initiated after a delay of T_3 from the initiation of τ_{3b1}.

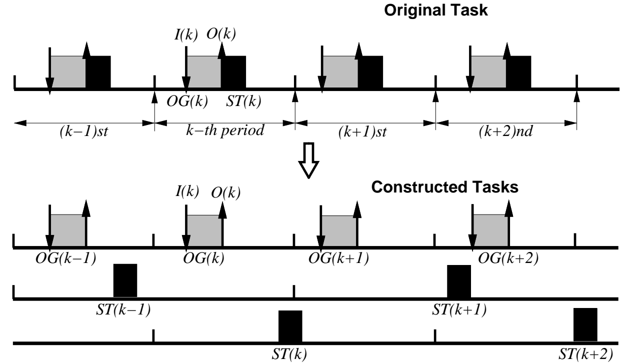


Figure 7: Scheduling of Newly Constructed Tasks.

Figure 7 shows a possible execution of the three new tasks $\{\tau_{3a}, \tau_{3b1}, \tau_{3b2}\}$ after the task transformation.

Unfortunately, *even this* transformation is unsafe, unless we ensure that the precedence constraints between the subtasks are maintained. To name just two, the ith instance of τ_{3b1} must finish before the ith instance of τ_{3b2}; or, the kth instance of τ_{3b1} must finish before the $2k+1^{st}$ instance of τ_{3a} starts. However, we present in Section 3.4 a simple (dynamic) modification to rate-monotonic scheduling that maintains our desired precedence constraints. Moreover, the modification preserves standard rate-monotonic analysis; that is, critical-instant analysis suffices to show schedulability. With this fact, we now analyze the schedulability of the newly constructed task set.

Example 1 (revisited): Consider the constructed task set $\{\tau_1, \tau_2, \tau_{3a}, \tau_{3b1}, \tau_{3b2}\}$. For the sake of schedulability analysis, we can coalesce τ_{3b1} and τ_{3b2} back into τ_{3B}, since they have the same period. (Of course, $T_{3B} = 2T_3$ and $C_{3B} = C_{3b1} + C_{3b2}$.)

The first two tasks are schedulable, as we have shown in Example 1. The first three tasks including τ_{3a} can be shown to be schedulable at scheduling point T_3. Finally, the entire task set is schedulable, since it passes the schedulability test at scheduling point $3T_2$.

$$\begin{array}{l} 3C_1 + 2C_2 + C_{3a} \le T_3 \\ \qquad (12 + 8 + 4.93 < 25) \\ 5C_1 + 3C_2 + 2C_{3a} + C_{3B} \le 3T_2 \\ \qquad (20 + 12 + 9.86 + 3.04 < 48) \end{array}$$

□

As long as the precedence constraints are maintained, the above transformation guarantees that observable operations meet their deadlines.

We now present the task transformation algorithm in Figure 8. As input, the algorithm is given a task set Γ, in which tasks are ordered in the increasing order of their periods. Here, $|\Gamma|$ is the cardinality of the set Γ and τ_k denotes the kth task in Γ. Mark$[\tau]$ is a flag

associated with τ, which is used to indicate that task τ is not transformable when the flag is checked ($\surd$).

The function *MinPoint*(k,T) calculates the overload associated with all the scheduling points, and selects the point where the overload achieves a minimum. This point – as well as the overload associated with it – is returned by the function.

The transformation algorithm itself examines tasks in Γ in order. For each task τ_k, it calls *MinPoint*(k,T_k) to compute the excessive processor demand (line (8)). If there is no excessive demand, this means that the first k tasks are schedulable (line (9)), and the algorithm goes on to the $k+1^{st}$ task.

At this point τ_k is deemed to be unschedulable, so the algorithm attempts to transform it. If it is found to be marked – that is, the state-computation element of a previously decomposed task – it cannot be transformed further. Thus the algorithm exits. Otherwise, the scheduling point "p" in [0, $2T_k$] is retrieved by again calling *MinPoint*(k,$2T_k$) (line (11)). Then the algorithm determines the idle time (t_{spare}) in the interval [0, p] (line 12).

If the amount of the overload is greater than the available idle time, schedulability is not achieved (line (13)). There is hope, however, if sufficient delay exists. In this case the algorithm calls a subroutine (**Algorithm 4.2**) to actually produce specialized subtasks as in Figure 6. In Section 4 we present the subroutine, as well as the method of program-slicing on which it is based.

The subroutine produces the two residual threads of τ_k, τ_{ka} and τ_{kb}, and then two copies of τ_{kb} are coalesced to produce τ_{kB}. (Recall that $C_{kB} = C_{kb1} + C_{kb2}$ and that $T_{kB} = 2T_k$.) On line (15), the excessive processor demand at τ_k's critical instance is reduced. Line (16) checks the sufficiency of the transformation. If it was successful, the two threads are marked and placed into the list Γ in their appropriate places. Thus Γ may grow to a maximum of twice its original size. Otherwise, the task set is deemed unschedulable.

Of course, the algorithm can be enhanced by allowing decomposition of tasks $\{\tau_1, \tau_2, \ldots, \tau_{k-1}\}$, if the decomposition of τ_k is determined to be insufficient on line (16). However, this alteration requires an additional loop as well as a more complex control structure. For the sake of brevity, we do not present the enhanced algorithm here.

```
(1)   function MinPoint( k, T )
(2)   let f(j,t) = sum_{i=1}^{j} C_i ⌈t/T_i⌉ − t;
      begin
(3)     SP := {j · T_i | 1 ≤ i ≤ k, j = ⌊T/T_i⌋};
(4)     Find s ∈ SP such that f(k,s) = min{t ∈ SP | f(k,t)};
(5)     return (s, f(k,s));
      end

(6)   algorithm Transform Task Set Γ
      begin
(7)     for k := 1 to |Γ| do
(8)       ( , t_excess) := MinPoint(k, T_k);
(9)       if t_excess > 0 then begin
(10)          if Mark[τ_k] = √ then exit(failure);
(11)          (p, t) := MinPoint(k, 2 * T_k);
(12)          t_spare := −t;
(13)          if t_excess > t_spare then exit(failure);
(14)          Decompose task τ_k into τ_ka and τ_kB
                  using Algorithm 4.2;
(15)          t_excess := t_excess − (C_k − C_ka);
(16)          if t_excess > 0 then exit(failure);
(17)          Mark[τ_ka] := Mark[τ_kB] := √;
(18)          Replace τ_k with τ_ka in Γ;
(19)          Insert τ_kB in Γ with its period as T_kB;
          end
        end
      end
```

Figure 8: Algorithm for Task Transformation.

3.4 Modifying the Scheduler: Priority Exchange

As we have stated, deadline postponement assigns to the rate-monotonic scheduler the duty of ordering subtasks, so that they observe the precedence constraints introduced earlier in Section 3.1. However, a "pure" rate-monotonic scheduling algorithm is unable to enforce these constraints, except for simple precedences between high-rate tasks and low-rate tasks. This mandates that we modify the scheduling algorithm.

Before we proceed, we rewrite below the original precedence constraints in terms of three subtasks $\{\tau_a, \tau_{b1}, \tau_{b2}\}$, which are constructed from the task τ. Recall that τ_{b1} and τ_{b2} are copies of τ_b, which result from applying deadline postponement to the state-update component of τ. Here, we let τ^k denote the instance of task τ at the kth period.

$$\text{(C1)}\ \tau_{b1}^{k} \prec \tau_{b2}^{k} \quad \text{and} \quad \text{(C2)}\ \tau_{b2}^{k} \prec \tau_{b1}^{k+1}$$
$$\text{(C3)}\ \tau_{a}^{2k} \prec \tau_{b1}^{k} \quad \text{and} \quad \text{(C4)}\ \tau_{a}^{2k+1} \prec \tau_{b2}^{k}$$
$$\text{(C5)}\ \tau_{b1}^{k} \prec \tau_{a}^{2k+1} \quad \text{and} \quad \text{(C6)}\ \tau_{b2}^{k} \prec \tau_{a}^{2(k+1)}$$

The rate-monotonic scheduling algorithm can easily ensure constraints C1 and C2 via a tie-breaking scheme, in which when two tasks have the same priority, the task with an earlier deadline (also an earlier period start time) is favored. Also, it guarantees constraints C3 and C4, since τ_a has a higher priority than

both τ_{b1} and τ_{b2}. However, a pure rate-monotonic priority assignment cannot guarantee constraints C5 and C6 without sacrificing schedulability. Such a case happens when task τ_{b1} (or τ_{b2}) has not finished during the first half of its period. This problem leads to a dynamic modification to rate-monotonic scheduling called *priority exchange*.

The priority exchange scheme enables two phased, harmonic tasks to preserve our necessary precedence constraints during their execution. We present the mechanism with the tasks τ_a and τ_{b1} where p_a and p_{b1} respectively denote their priorities.[2]

- When a period of τ_a starts in the middle of T_{b1}, and if τ_{b1} has not yet finished its execution, then τ_{b1} exchanges its priority with τ_a. Also, a countdown timer gets set to C_a.
- The timer is only decremented (1) if it has been set, *and* (2) if τ_{b1} or τ_a are *running* with priority p_a. That is, if either τ_{b1} or τ_a get preempted by a higher priority task, the timer is temporarily stopped.
- If τ_{b1} finishes *before* the timer expires, then τ_a is restored to its original priority p_a.
- When the timer expires, the currently running task gets p_{b1} as its priority, and proceeds.

The last case induces a subtle situation, which deserves further explanation. Consider a scenario in which the timer expires while τ_{b1} is running. According to the protocol, τ_{b1} is restored to its original priority of p_{b1}, and it proceeds. Then, at some time after τ_{b1} finishes, both τ_a and τ_{b2} may be in the ready queue. Constraints C3 or C4 mandate that τ_a executes first. However, the protocol achieves the desired effect: although τ_a inherited the priority of τ_{b1}, by the tie-breaking scheme τ_a is dispatched first.

The implementation of the priority exchange scheme requires one countdown timer for each "family" of slices, e.g., τ_a, τ_{b1} and τ_{b2}. However, the countdown timers for all such "families" of tasks can easily be managed by only one programmable hardware timer – a standard component in most systems. Also, the priority exchange scheme can easily be achieved by a slight modification to the kernel.

Our task transformation tool automatically extracts the needed precedence information for the runtime system. Most importantly, priority exchange allows us to continue utilizing standard rate-monotonic schedulability, while guaranteeing the desired precedence constraints.

[2] The same mechanism is applied to arbitrate between τ_a and τ_{b2}.

4 Automatic Task Decomposition by Program Slicing

The idea behind task decomposition is to accept a task, and then generate its two code components as discussed in Section 3.1. That is, one thread triggers all observable events, while the other computes the next-state update. Straightforward as it may look, the decomposition can in reality be a very complex compiler problem. Many factors make this the case, among which are intertwined threads of control, nested control structures, complex data dependences between statements, procedure calls in the task code, etc. To cope with these problems in a systematic manner, we harness a novel application of *program slicing* [19, 24, 25]. For the sake of brevity, we assume the following:

- Function calls are inlined.
- Loops are unrolled.
- The intermediate code of programs is translated into static single assignment form [2, 7].

The first assumption allows us to avoid *interprocedural* slicing [8]. The next two assumptions simplify problems induced by spurious data dependences such as anti-dependences and output dependences [1]. However, we can easily alleviate the restrictions, relying on dependence breaking transformations, such as scalar expansion [1]. Static single assignment is one such transformation.

A *slice* of program P with respect to program point p and variable v consists of P's statements and control predicates that may affect the value of v at point p. We call a pair $\langle p, v\rangle$ a *slicing criterion*, and denote its associated slice by $P/\langle p, v\rangle$. The result is that we can just execute the slice $P/\langle p, v\rangle$ to obtain the value of v at location p. Recall our periodic controller task of Figure 5, which we call P_{control}. The following fragment is the slice $P_{\text{control}}/\langle eot, \texttt{state}\rangle$ where eot is a pseudo-location at the end of the loop body.

```
      every 25ms
        {
L1:     receive(Sensor, data);
L2:     if (!null(data))
          {
L3:       t1 = F1(state);
L4:       t2 = F2(state);
L5:       t3 = F3(data);
L9:       state = t1 * ( t2 + t3 );
          }
        }
```

Statements L1, L3, L4, L5 and L9 are included in the slice, because variable "**state**" depends on their computations (this is called *data dependence*). Also, the predicate on line L2 is included, because the execution of statements L3, L4, L5 and L9 (hence the value of "**state**") depends on the boolean outcome of the predicate (this is called *control dependence*).

Thus the computation of slices is based on data dependence as well as control dependence. In this regard, using a *program dependence graph* [4, 8, 19] is ideal, since it represents both types of dependences in a single graph. The program dependence graph is defined as follows.

Definition 4.1 The program dependence graph is a directed graph $G = (V, E)$, where

- The vertices V represent the task's statements; i.e., assignments, control predicates and observable statements (such as **send** and **receive**). In addition there is a distinguished vertex "entry," which represents the root of the task.
- The edges E are of two sorts. An edge $n_1 \xrightarrow{c} n_2$ denotes a control dependence between n_1 and n_2. That is, either (1) n_1 is an entry vertex and n_2 is not nested within any loop or conditional, or (2) n_1 represents a control predicate and n_2 is immediately nested within the loop or conditional whose predicate is represented by n_1. An edge $n_1 \xrightarrow{d} n_2$ denotes a data dependence. That is (1) n_1 defines variable v, and n_2 uses v, *and* (2) control can reach n_2 after n_1 via an execution path along which there is no redefinition of v.

In general, we can further classify $n_1 \xrightarrow{d} n_2$ as either *loop independent* or *loop-carried.* A data dependence is carried by a loop if the execution path between n_1 and n_2 includes a backedge to the loop header, and both n_1 and n_2 are enclosed within the loop.

Finally, we define "$p \Rightarrow_* q$" to mean that node p can reach node q via zero or more control dependence edges or data dependence edges. □

The program dependence graph of our controller program is shown in Figure 9. Note that loop-carried dependences caused by "**state**" exist in the program dependence graph, even if there is no inner loop in the program: they derive from a loop-like nature of periodic tasks. We call these dependences *periodic loop-carried* dependences.

The slice of program P with respect to program point p and variable v (i.e., $P/\langle p, v\rangle$) can be obtained through a traversal of P's program dependence graph. A simple algorithm to compute the slice is given below.

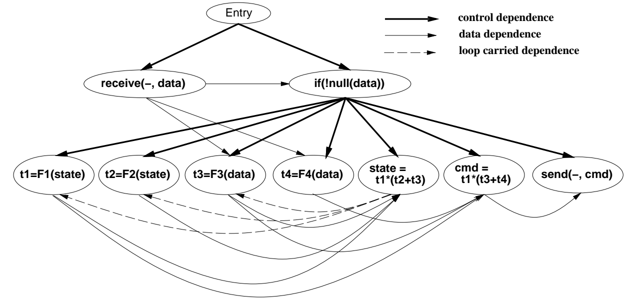


Figure 9: Program Dependence Graph.

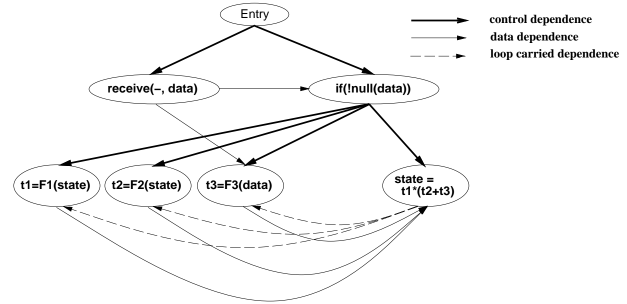


Figure 10: Slice of Program Dependence Graph.

In the algorithm the program point p corresponds to a vertex of G.

Algorithm 4.1 *Computes the slice $P/\langle p, v\rangle$:*

Step 1 Compute reaching definitions $RD(p, v)$ such that for any vertex $n \in RD(p, v)$, the statement n defines variable v, and control can reach p from n via an execution path along which there is no redefinition of v.

Step 2 Compute the slice by a backward traversal of G such that
$P/\langle p, v\rangle = \{m \mid \exists n \in RD(p, v) \ : \ m \Rightarrow_* n\}$.

The definition of the program slice can be extended for a set of slicing criteria C in such a way that $P/C = \bigcup_{\langle p,v\rangle \in C} P/\langle p, v\rangle$.

Figure 10 shows the graph that results from taking a slice of the program dependence graph in Figure 9 with respect to criterion $\langle eot, \texttt{state}\rangle$, where eot is an imaginary vertex representing the end of the task.

The most essential part of our task decomposition algorithm is to pick the right slicing criteria so that the resulting slices of a task "cover" all behaviors of the original task. To meet the requirements of the algorithm in Figure 8, which drives the decomposition, we use the two following sets of slicing criteria for task τ.

(1) Set $C_o(\tau)$ of all slicing criteria $\langle o, var(o)\rangle$ where o is an observable operation which occurs in task

τ's code, and $var(o)$ is a variable[3] appearing in o.

(2) Set $C_s(\tau)$ of slicing criteria $\langle eot, s \rangle$ where s is a state variable in τ. In a periodic task, a state variable is defined to be a variable which causes a periodic loop-carried dependence.

From the viewpoint of our application domain, this decomposition is correct if the two slices $C_o(\tau)$ and $C_s(\tau)$ preserve the task τ's original behavior. In other words, if the two slices are scheduled to maintain their precedence constraints, then for any initial state s, all original observable behaviors are exhibited by $C_o(\tau)$ and $C_s(\tau)$. The following facts ensure that this is true:

- Variables defined in task τ can be classified into two classes: (1) variables which affect observable output operations through data dependences, periodic loop-carried dependences and/or control dependences; and (2) variables which do not affect output.
- All statements defining class (1) variables are included in τ/C_o or τ/C_s.
- All statements defining a type (2) variable can be deleted, since they do not change the original observable behaviors.

Using the two criterion sets, the task decomposition algorithm is given below:

Algorithm 4.2 *Decompose task τ into τ_a and τ_b:*

Step 1 Compute $C_o(\tau)$ and slice task τ with respect to $C_o(\tau)$. Then the generated slice $\tau/C_o(\tau)$ becomes τ_a.

Step 2 Compute $C_s(\tau)$ and slice task τ with respect and $C_s(\tau)$.

Step 3 Delete from $\tau/C_s(\tau)$ non-conditional statements common to both of the slices. The remaining code becomes τ_b.

Recall the source code of τ_3, as shown in Figure 5. Establishing our slicing criteria, we find that $C_o(\tau_3) = \{\langle \text{L1}, \texttt{data} \rangle, \langle \text{L8}, \texttt{cmd} \rangle\}$, and $C_s(\tau_3) = \{\langle \text{L10}, \texttt{state} \rangle\}$. Figure 6 shows the slice $\tau/C_o(\tau)$ as subtask τ_{3a}. The original slice $\tau/C_s(\tau)$ was given earlier in this subsection as an introductory example, while the subtask τ_{3b} in Figure 6 shows the final result of removing common statements L1 and L3 from $\tau/C_s(\tau)$.

[3] For notational convenience, we assume that each observable operation has at most one variable.

5 Conclusion

In this paper we have presented (1) a new real time programming language, TCEL, whose semantics is based on the interrelationships between observable events; and (2) a compilation technique which automates task tuning operations for enhanced schedulability.

There have been several other compiler-based approaches to real-time programming. In [22] a schedulability analyzer is embedded in a compiler to extract and analyze timing information from the assembly-language output. In [6] a compiler classifies application code on the basis of its predictability and monotonicity, and creates partitions which have a higher degree of adaptability. In [18] a partial evaluator is applied to a source program, which produces residual code that is both more optimized and more deterministic. In [17] time-critical statements (or events) are assumed in the underlying programming language, and used for developing the notion of safe real-time code transformations. Based on this notion of safety, a large number of conventional code transformations are examined, and then classified for application in real-time programming. Finally, in [7] a code scheduling algorithm is presented, which is used to achieve the timing consistency of sequential real-time programs. Each of these efforts addresses a different problem associated with real-time programming.

The work in [17] comes closest to this work, in that the timing behavior of real-time programs is described in terms of events or the executions of time-critical statements. Unlike the semantics for TCEL, however, the execution times of non-time-critical statements are not explicitly decoupled from timing constraints imposed on the events. Thus, the applicability of some transformations may be unnecessarily restricted.

The compiler-assisted scheduling approach in [6] is also similar to this work, in that the compiler creates code partitions to mainly support run-time adaptability of a real-time scheduler. However, the partitioning is guided by "hints" supplied by the programmer. Further, it is limited to code segment breaking. Our transformation, on the other hand, uses static dataflow analysis to isolate intertwined threads of control.

Also, the partial evaluator in [18] is tangentially related to program slicing, in that both techniques specialize programs. However, our approach also *decomposes* programs – producing multiple specialized versions for different variables and inputs. In this manner, the program slicer can adapt to a given scheduling technique. The partial evaluator, on the other hand, manages problems such as program analysis and eval-

uation.

While our focus here is on rate monotonic scheduling, our transformation techniques are not necessarily confined to a particular scheduling paradigm. For example, task decomposition by program slicing can allow the state-update subtask to fill the delay which occurs before a "**send**" operation in a periodic task. This transformation can optimize out waiting time due to a synchronization requirement and thus, utilize the potentially wasted cycles. This, in turn, may enhance the schedulability of the entire task set. In the same manner, our semantics-based compiler transformations will be even more beneficial for schedulers that handle periodic tasks with inserted offsets [23].

References

[1] F. Allen, B. Rosen, and K. Zadeck. *the forthcoming Optimization in Compilers*. Addison Wesley Publishing Company, 1992.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and systems*, 9:319–345, July 1987.

[3] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, method for validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, January 1985.

[4] J. Ferrante and K. Ottenstein. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and systems*, 9:319–345, July 1987.

[5] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computer*, 30:478–490, July 1981.

[6] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 247–256. IEEE, December 1990.

[7] S. Hong and R. Gerber. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM Press, June 1993. *SIGPLAN Notices, 28(6):166-176*.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graph. *ACM Transactions on Programming Languages and systems*, 12:26–60, January 1990.

[9] Y. Ishikawa, H. Tokuda, and C. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of OOPSLA-90*, pages 289–298, October 1990.

[10] F. Jahanian and Al Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[11] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12:941–949, September 1986.

[12] J. Krause. GN&C domain modeling: Functionality requirements for fixed rate algorithms. Technical Report (DRAFT) version 0.2, Honeywell Systems and Research Center, December 1991.

[13] I. Lee and V. Gehlot. Language constructs for real-time programming. In *Proceedings IEEE Real-Time Systems Symposium*, pages 57–66. IEEE, 1985.

[14] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171. IEEE, December 1989.

[15] K. J. Lin and S. Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proceedings IEEE Real-Time Systems Symposium*. IEEE, December 1988.

[16] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[17] T. Marlowe and S. Masticola. Safe optimization for hard real-time programming. In *Second International Conference on Systems Integration*, pages 438–446, June 1992.

[18] V. Nirkhe. *Application of Partial Evaluation to Hard Real-Time Programming*. PhD thesis, Department of Computer Science, University of Maryland at College Park, May 1992.

[19] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[20] C. Park and A. Shaw. Experimenting with a program timing tool based on source-level timing schema. In *Proceedings IEEE Real-Time Systems Symposium*, pages 72–81. IEEE, December 1990.

[21] L. Sha, J. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings IEEE Real-Time Systems Symposium*, pages 181–191. IEEE, December 1986.

[22] A. Stoyenko. A schedulability analyzer for real-time Euclid. In *Proceedings IEEE Real-Time Systems Symposium*, pages 218–227. IEEE, December 1987.

[23] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS 182 (1992), Department of Computer Science, University of York, England, August 1992.

[24] G. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[25] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[26] V. Wolfe, S. Davidson, and I. Lee. RTC: Language support for real-time concurrency. In *Proceedings IEEE Real-Time Systems Symposium*, pages 43–52. IEEE, December 1991.

[27] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *The Journal of Real-Time Systems*, 5(4), October 1993. To appear.